ROKOKO

An Introduction to Unity Animator Controller

A step-by-step walkthrough of practical workflows for gamedev character animation.

Introduction: **2**

1.0 Setting up: **3**
1.1 Choosing the right version of Unity: 3
1.2 Choosing a directory that make sense: 4
1.3 Choosing the right pipeline: 5
1.4 Changing the dang light-bulb: 6
1.6 Organizing your layout: 6
1.5 Setting up your scenes and folders: 6

2.0 Importing: **8**
2.1 Identify dependencies: 8
2.2 Separate material from fbx: 9
2.3 Create appropriate folders (again): 11
2.3 Create and save character as prefab: 12
2.4 Bonus practice: 15
2.5 Useful links (optional): 16

3.0 Retargeting: **16**
3.1 Process summary: 17
3.2 Import fbx with animation: 17
3.3 Converting animation to humanoid: 19
3.4 Renaming and sorting your animation clips: 25
3.4.1 (OPTIONAL) Duplicating the animation clip: 26
3.5 Convert character rig to humanoid avatar 27
3.5.1 Bonus practice (hand surgery): 31

4.0 Animation basics: **32**
4.1 Testing and modifying animation clips 33
4.1.1 The settings: 35
4.1.2 Making modifications 38
4.1.3 Modifying animation rest poses: 42
4.1.4 (BONUS) Making multiple clips from one FBX-file: 45
4.2 The Animator Component (and Animation Controller) 46
4.2.1 The Animator Component 46
4.2.2 The Animation Controller 48
4.2.2.1 Layers (basics): 50
4.2.2.2 Parameters: 50
4.2.2.3 Layer Editor: 50
4.3 Building the state machine: 53
4.3.1 State transition setup: 54
4.3.2 Writing the basic logic: 58
Declarations: 59
Start function: 59
Update function: 60

4.4 Blend trees: 60
4.4.1 Driving blend trees: 70
4.4.2 (BONUS) Root transform corrections: 74
4.5 Face Capture - Implementing blendshape animations: 76
4.5.1 Renaming the mesh 81
4.6 Playing multiple skeletal animations simultaneously: 91
4.7 Mocap cleanup (Blender, mostly): 95
4.7.1 Removing jitter: 95
4.7.2 Doing pose corrections: 102

5.0 Rounding off: 104

6.0 Where to go next: 105
6.1 Rokoko Discord Community 105
6.2 Online tutorials 105
6.3 Asset resources 106

Introduction:

Hello! My name is David Lindberg, I work for Rokoko as a Technical Community Manager. While I have a lot of prior experience in Unity, I had done very little with their animation systems until now. This document chronicles my learning process, to create what is hopefully a good introduction for anyone who is also just beginning to learn about animations in Unity.

This document aims to introduce the many tools and techniques that are available in Unity for animating characters with exported animations. **This article *does not cover livestreaming from Rokoko Studio into Unity*, as I believe that deserves its own article, addressing different challenges and workarounds in that space.**

There are a few layers of stuff to go through. The most basic outline is as follows:

Content:
- The animation clip itself (mocap or hand animated)
- The character model (w. Skeleton, mesh, materials, textures)

Tools:
- The avatar system (retargeting between humanoid rigs)
- The animation timeline (preview/edit/add logic)
- The animation controller (state machine)

Knowing how to use these tools to manipulate the content is a really good starting point, from which we can expand into more complex systems and scenarios.

There's of course an endless assortment of other plugins and cool third-party stuff that we can implement in Unity, but for this article we will cover what has been developed by Unity themselves. As an aside, while we will be going through the basics, I recommend people who have never used Unity before to go check out their learning path for beginners first. It isn't mandatory, but I reckon it will help, especially for navigating the interface:

- https://learn.unity.com/pathway/unity-essentials

Or alternatively, this awesome crash course by Imphenzia:

- LEARN UNITY - The Most BASIC TUTORIAL I'll Ever Make

Finally, please be aware that this is not a one-stop-solution for every humanoid animation onto any humanoid character. A lot of the time you'll go through these steps and get great results - yet oftentimes you'll also have to make iterations, corrections to ensure that everything lines up properly. But regardless of the challenges you face, the first step is to know what tools you have at hand and how they work, which is the point of these tutorials.

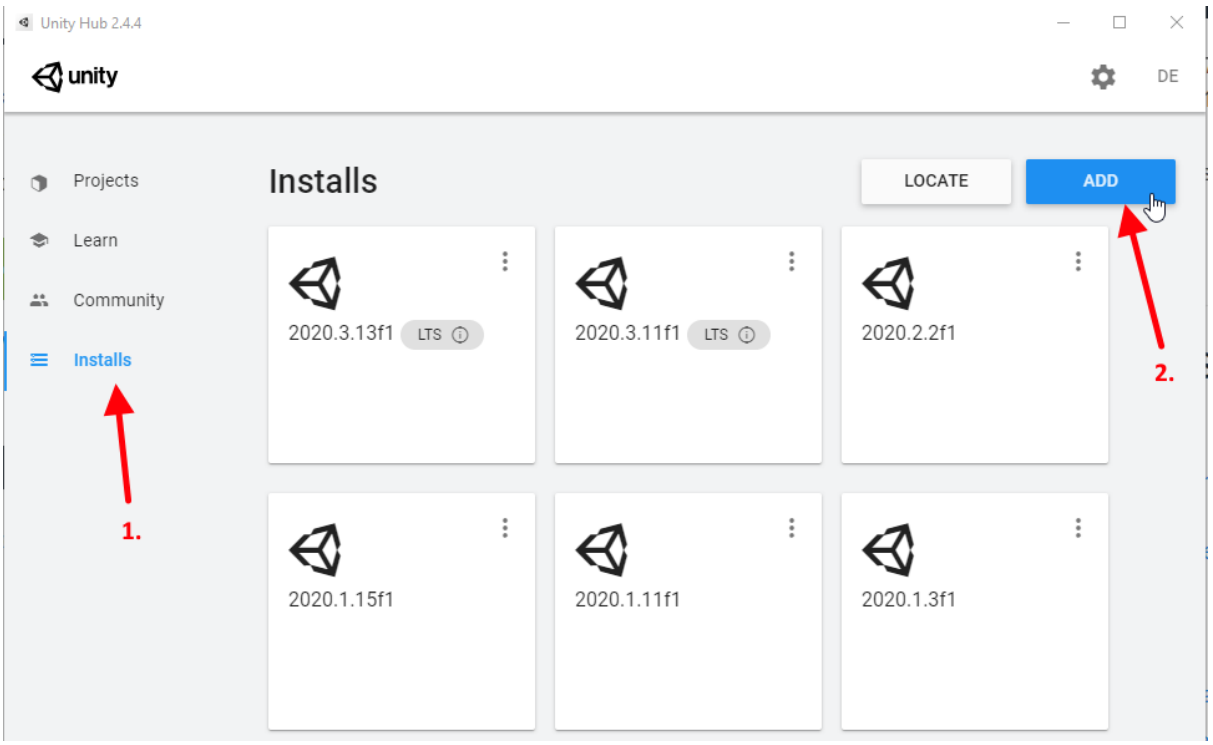
1.0 Setting up:

I am writing this introduction with the assumption that you have little to no experience with Unity, prior to this point. If you already have some experience with Unity, just skip ahead to section 2.0. When creating your Unity project, there's a few steps that can be handy to follow:

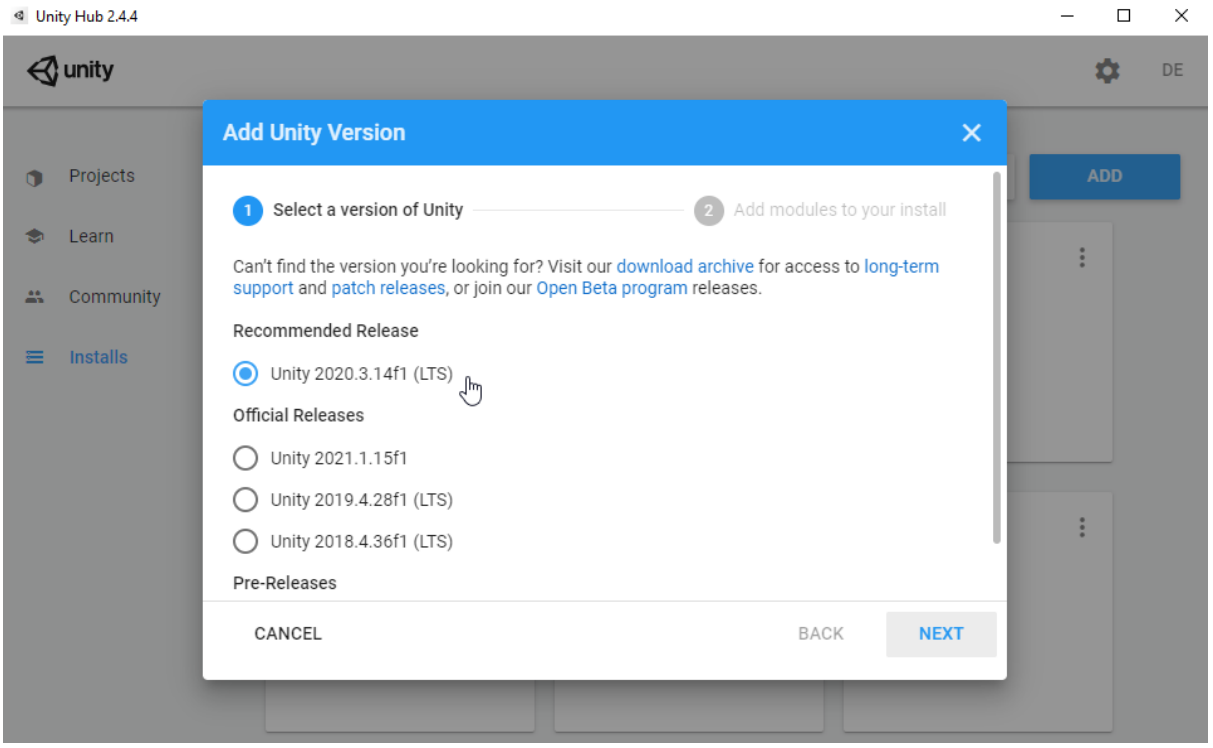
1.1 Choosing the right version of Unity:

In order to use Unity, we'll have to install Unity - And before we can install Unity, we need to install the Unity Hub, which helps us manage our installation process. This hub is useful due to Unity always getting updated with new features, meaning there are a bunch of versions to choose from. When you open the Hub;

1. *Click on 'Installs' to the left*
2. *Click the blue 'Add' button at the top right*:



My recommendation is to choose the most recent "Long Term Support (LTS)" version. Unity is usually showing this as the Recommended version:



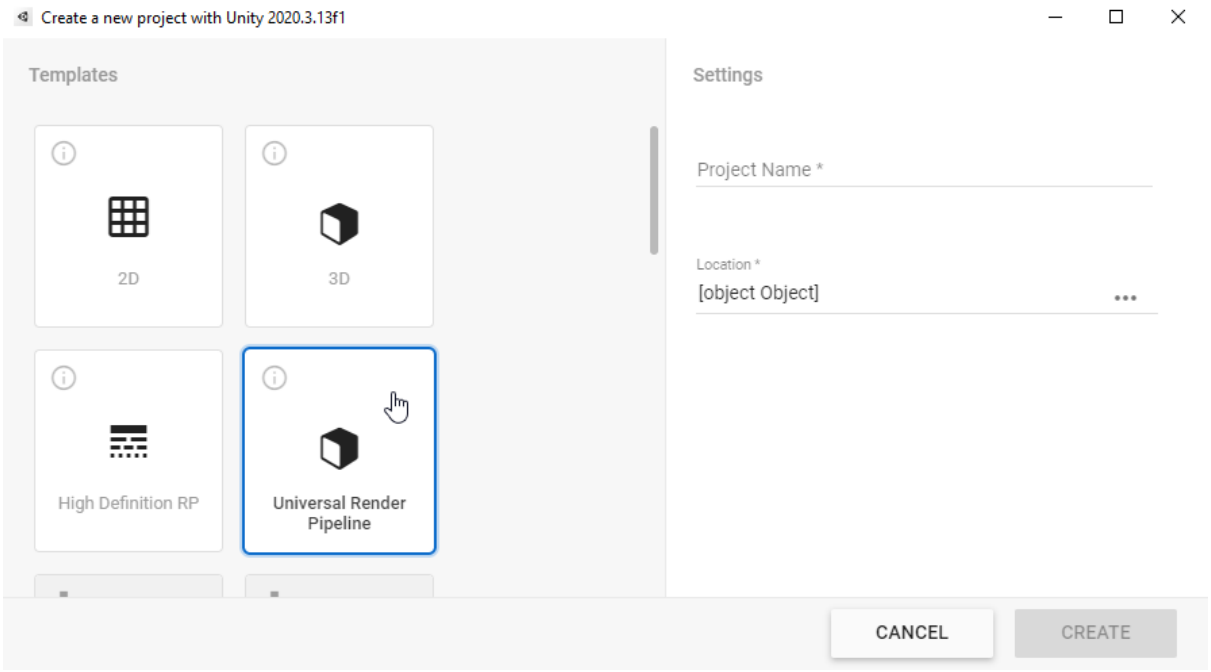
1.2 Choosing a directory that make sense:

This step is often a bit overlooked when making your first projects, but it's nice to have it managed from day one, once you start piling on more projects over time. You don't want to

start moving your directories around later. Just agree with yourself on where you want to store your files, preferably a location with space for lots of projects and assets.

1.3 Choosing the right pipeline:

A great thing about Unity is its flexibility. At its core it is very simple, with the idea being that you build or add your own modules (referred to as “packages”) to solve the problems that need to be solved. However, while this is great for people who know what they’re doing, it can be a little overwhelming for newcomers. Thankfully, to address this problem, Unity has some nice presets to choose from in the form of sample projects with preconfigured packages. These are visible when creating a new project, so go back to the **‘Projects’** tab on the left and click the blue **‘NEW’** button that has an arrow next to it (arrow is for selecting engine version to make the project in. Then you’ll be greeted by this view:



There’s a few game examples and stuff for VR and AR if you scroll down, but the most useful ones for the majority of cases, are the rendering pipelines:

Universal Render Pipeline (URP):
Used for projects where performance, wide platform support, and ease of customising graphics are the primary considerations.

High Definition RP (HDRP):
A good starting point for people focused on high-end graphics that want to develop games for platforms that support Shader Model 5.0 (DX11 and above).

For this tutorial I’d recommend URP, simply because it is more performant and you will likely use it in your own projects. Especially for anything stylized (eg. Inside, Valheim, Ori), the Universal Render Pipeline is your best bet.

1.4 Changing the dang light-bulb:

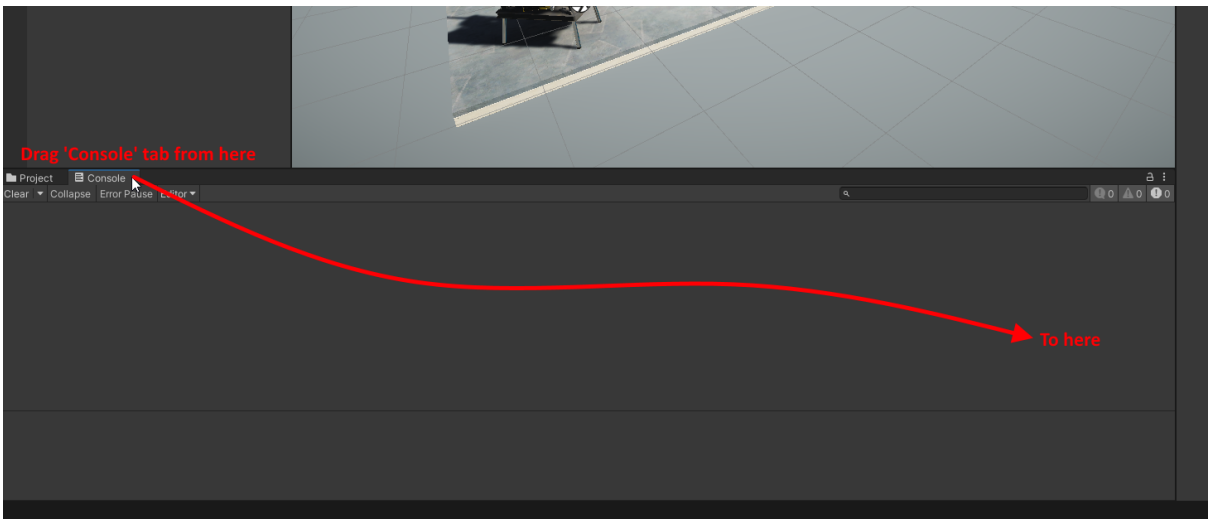
This is a bit of a pet-peeve of mine, but the default way that color is being calculated in Unity is a bit too shiny and plasticky, and not even more performant than its better looking counterpart. I **always** change this as the very first thing in a new project. From the top of the Unity editor, go to **Edit -> Project Settings**. This will open a new window:

1. Click '**Player**'
2. Expand the '**Other Settings**' tab
3. Change the color space from '**Linear**' to '**Gamma**'.

Unity will now ask you if you wish to change the color space to which you say yes please and thank you. And of course; [a more technical comparison of the two models], for those who are interested.

1.6 Organizing your layout:

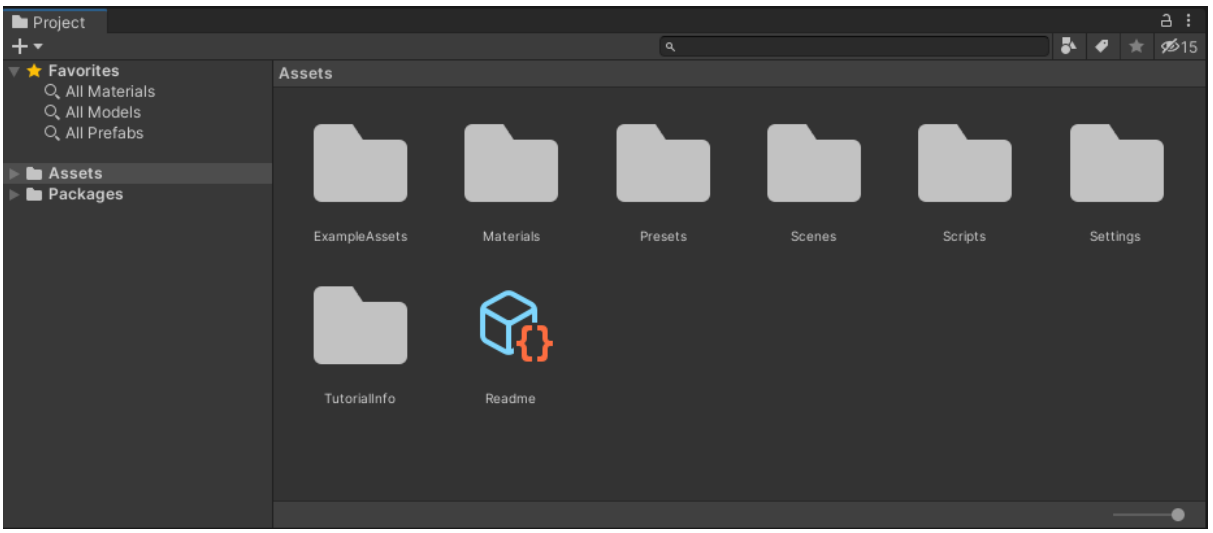
How you set up the layout of your Unity editor is generally a question of personal preference. However, if there is one thing I can generally recommend that you do, then it is to move the console window so that it is always visible. Even if you are not considering yourself a programmer - if something goes wrong in Unity, that's where the information goes. I like to do this:



That'll give your console some free real-estate next to the project window, instead of disappearing behind it.

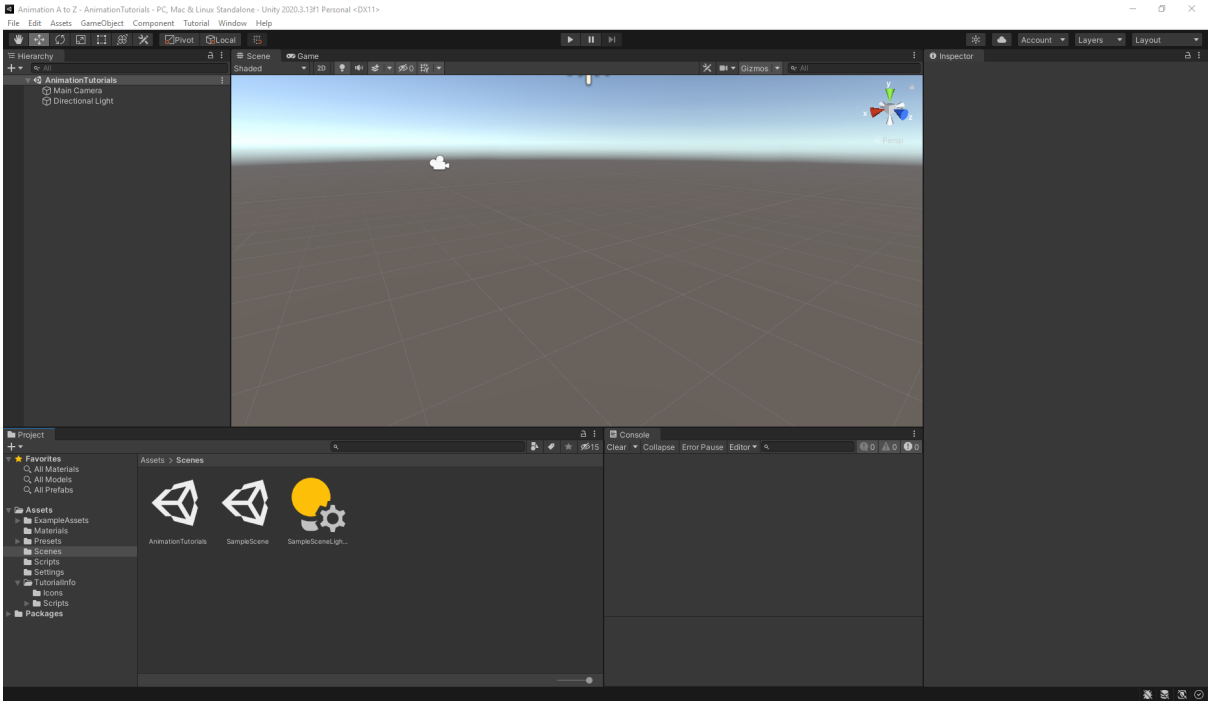
1.5 Setting up your scenes and folders:

You may have noticed that there are a lot of folders in your Unity project already. All of this comes from the URP package that was included when we made the project. The great thing about this, is that it demonstrates some general folder naming and sorting practices, which we should always strive to maintain, to prevent our projects from becoming unwieldy:



You can zoom in and out with the slider in the bottom right corner. Unfortunately the text is a bit smudgy on this picture, but if you look in your own project there's names like 'Materials', 'Scenes' and 'Scripts', which all indicate a specific type of content. Whenever you add something to your project, think about what you're adding and how you are going to easily find it if (when) you forget where you put it. As for the existing folders, just leave them be for now.

We're going to create a new scene for this tutorial anyway, so you might as well go into the existing 'Scenes' folder and create a new scene by right-clicking in the Project window and selecting **Create -> Scene**. I made one and named it 'AnimationTutorials' but you can call it whatever you want. Make sure you double-click it afterwards to open it up, which should leave you with an empty scene view like this:



2.0 Importing:

The first thing you want to do is to get the characters into your game. If you don't know where to start I will be using our Bruno character:

- https://cdn.rokoko.com/assets/unity/bruno.zip

And the viking Rømer:

- https://cdn.rokoko.com/assets/unity/romer.zip

You can use your own characters for this tutorial, but the great thing about starting with these two characters, is that each of them has some different limitations you'll likely run into now and again, so I'll be able to demonstrate how you can overcome these issues with each of them respectively. We won't cover the creation of the models themselves however, as that is for a different type of tutorial (and done in modelling software outside of Unity).

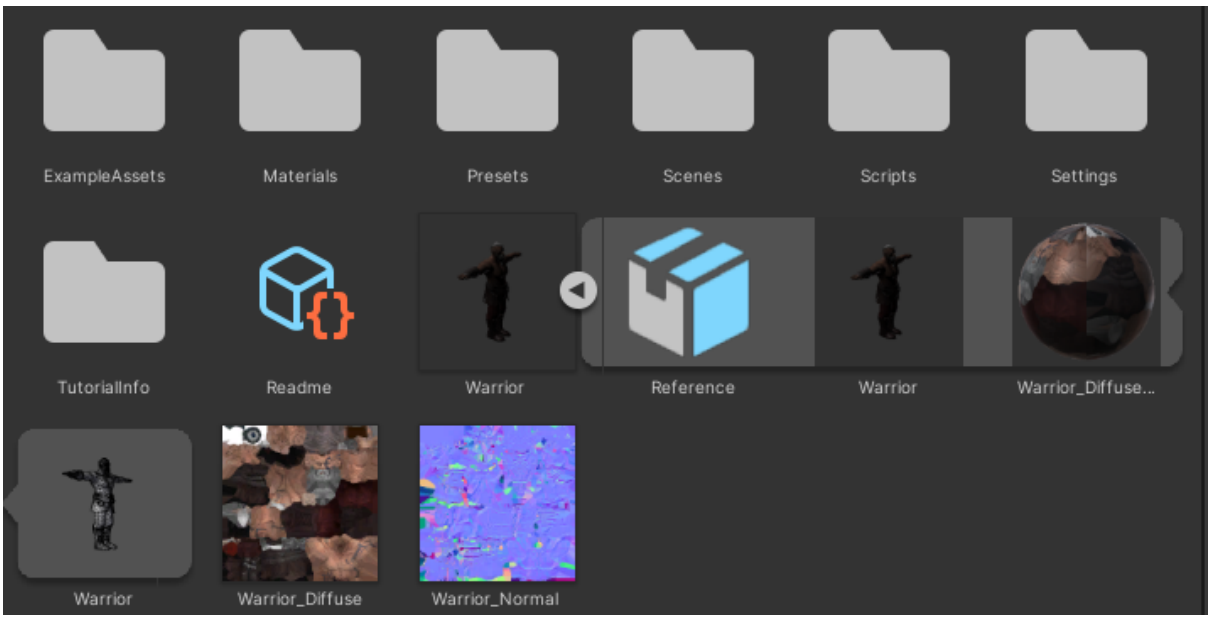
2.1 Identify dependencies:

If we start by unzipping the Rømer character folder, you'll notice that there are actually three files. One of them is the '**fbx**' (skeleton, mesh, materials and animation if any), the other two are textures:

The two textures here are:

- **Diffuse map:** The actual colours being displayed on the character.
- **Normal map:** Information describing how the surface is lit, allowing for additional surface detail without adding to the geometry.

Inside of this fbx-file, there's actually a material which makes use of these textures, so it is important that you drag all three files into Unity's project window at the same time, or poor Rømer will come out without any textures inside of Unity:

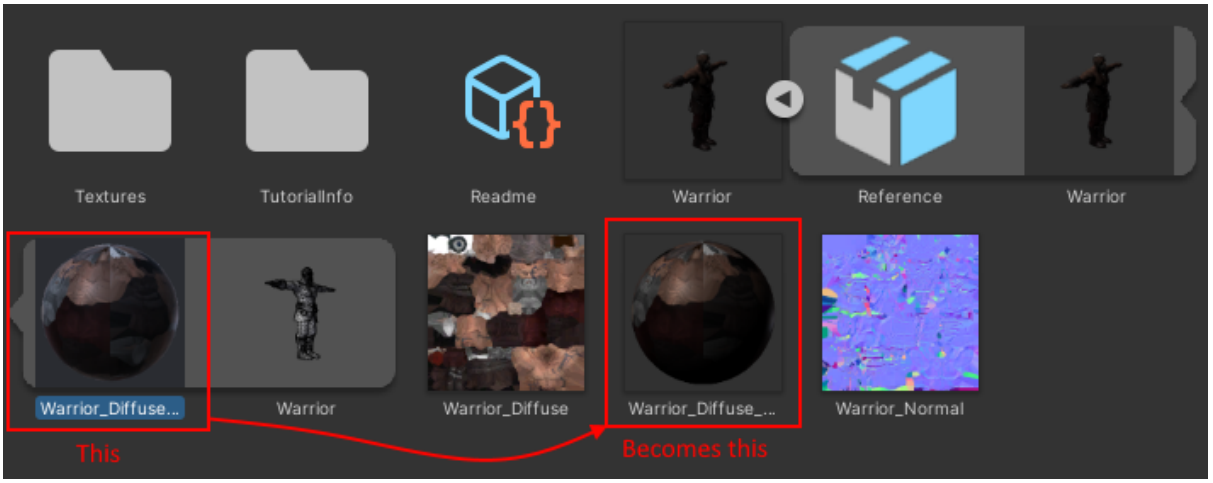


In this picture we see the fbx in a more 'unfolded' format inside of Unity, alongside the two textures. I won't go into too much detail about each of these components, other than the sphere you see here, which is the aforementioned material.

While it's nice to have the material contained inside of the fbx-file itself, it sometimes makes more sense to recreate it outside of the fbx, both for the sake of organizing your files but also to minimize situations where the read-only nature of the fbx might cause any problems, should you want to update the material at a later stage. This step is not mandatory, but might be helpful - especially if the original material was not saved correctly, as we will see below.

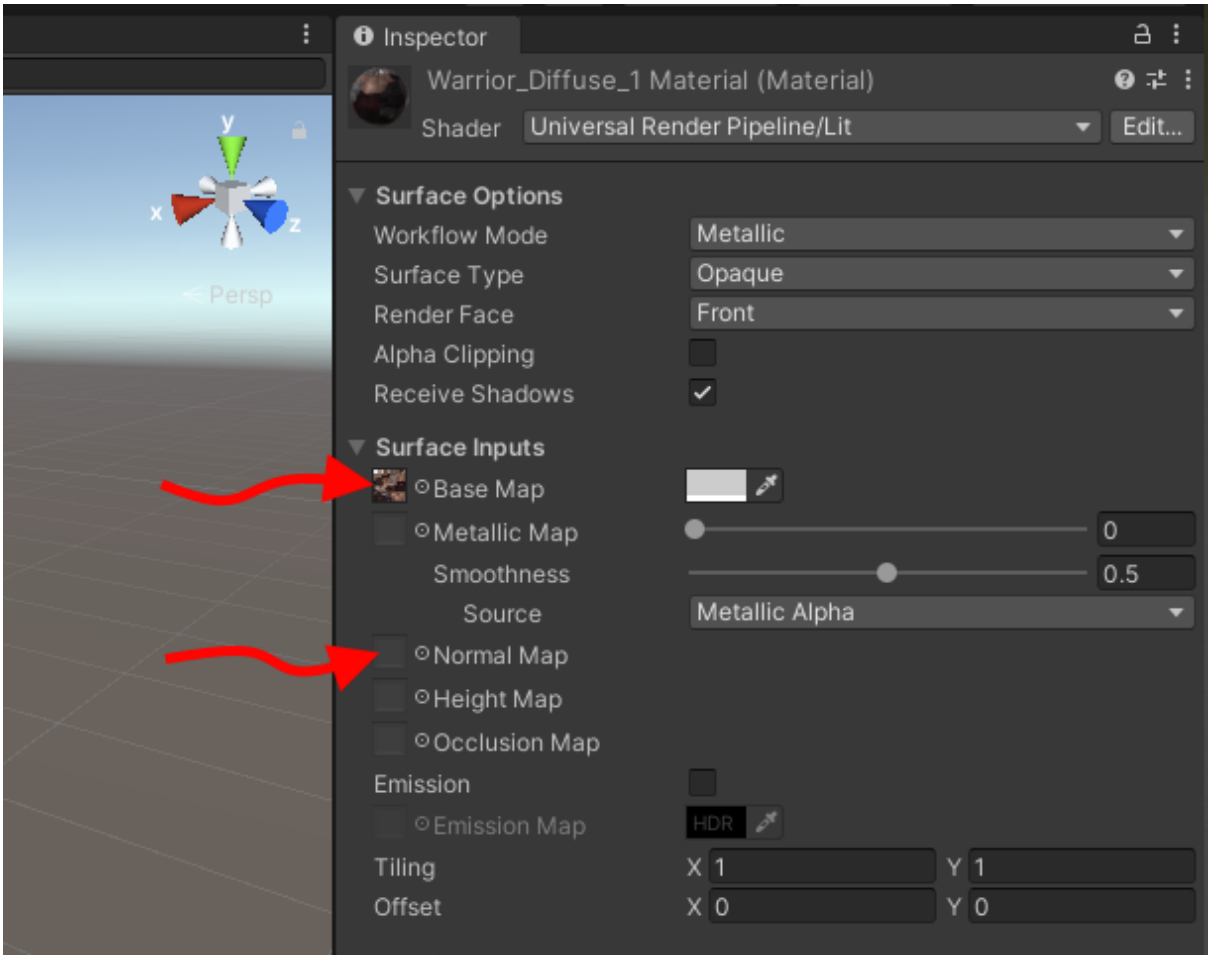
2.2 Separate material from fbx:

The easiest way to recreate the material outside of the fbx, is to select the material in the FBX and press CTRL+D to duplicate it:

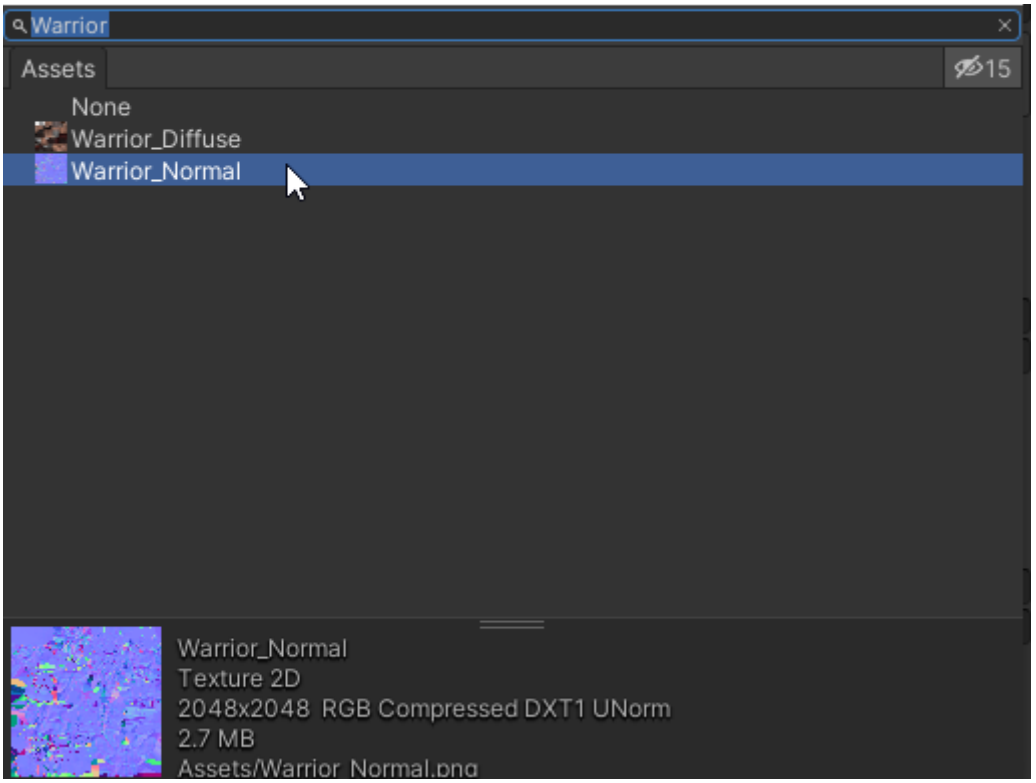


Let's take this opportunity to examine the material and see what it looks like, before we start using it. Click on the new material and check out the Inspector window to see the details. You'll see a lot of properties there, but we are only really concerned about the properties

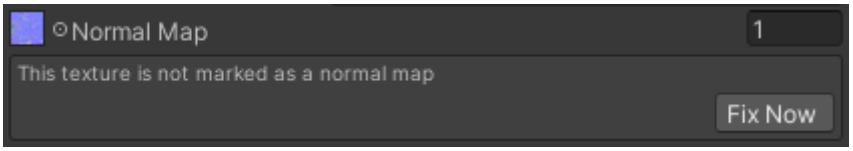
called **'Base Map'** (the diffuse texture) and the **'Normal map'** (the normal texture), both found under **'Surface Inputs'** if not currently visible:



A keen eye might realize at this point, that something is not quite right! Whoever saved the fbx-file originally forgot to actually apply the normal map, so it is not currently being used. Whoopsies! In order to apply it, **click the little circle-icon left of the** '**Normal Map**' and it'll open a file browser. Since we're using the basic URP project, it might be a little hard to spot the texture, as there's already a selection of them in the project. Since we know the name of the texture is "Warrior_Normal", we can simply find it by typing it into the search bar of the file browser, which will change the window from a tile view to a list view:



After you've selected the normal map, there's one last technical hiccup that should be addressed from the inspector when viewing the material. For the sake of handling everything correctly, Unity likes to know if a texture should be treated as a normal map (link for the curious ones). Since we've just thrown the texture into Unity without doing anything else, this warning will pop up in the Inspector, when applying it on our material:



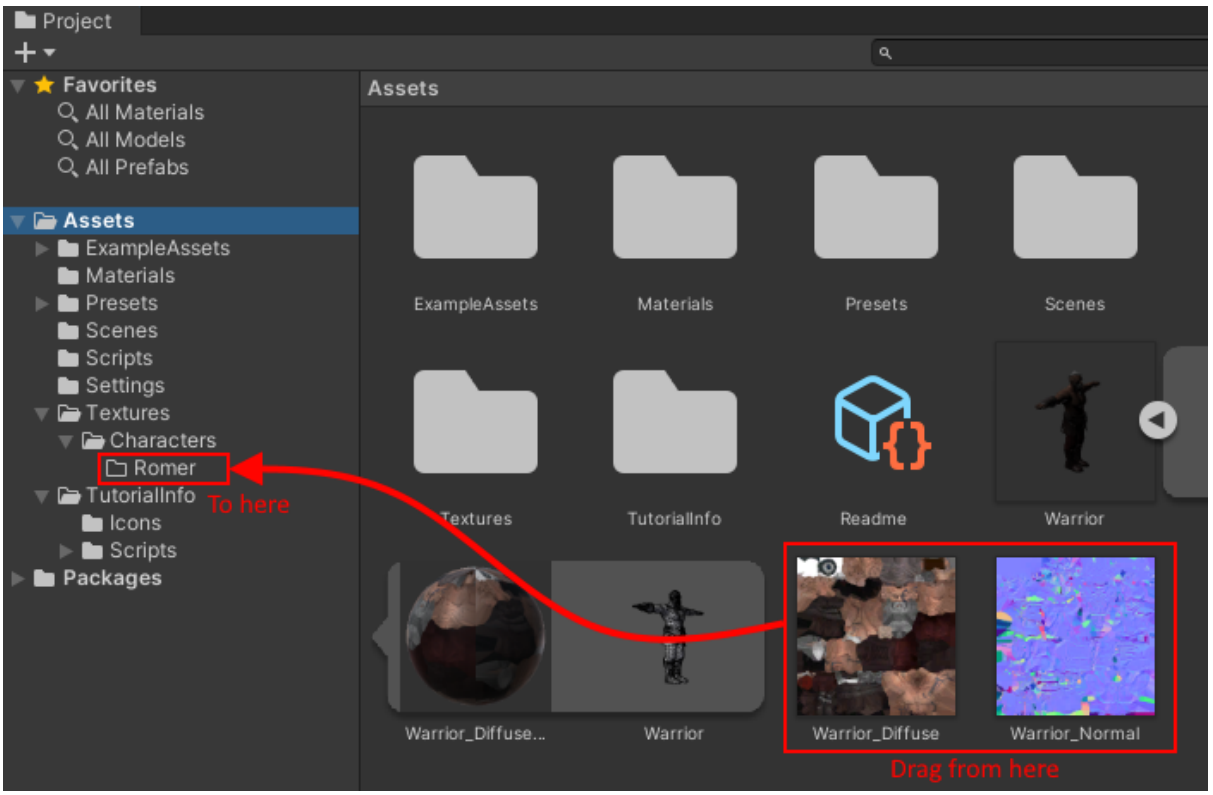
Just click the button!

Now you have an updated material that will not lose track of your textures, regardless of where you put them inside of your project.

2.3 Create appropriate folders (again):

I recommend making a **Textures** folder and dumping the textures in there. If you want to, you can create additional folders like **Characters** and then even one called **Romer** (not using the ø-letter as that can break stuff, but you can call it whatever you want).

An easy way to move stuff around in multiple layers of folders, is to use the asset browser on the right side of the Project window. Here I've made the folders I suggested and can now just drag and drop the textures via the browser:



CTRL-click or SHIFT-click to select multiple files

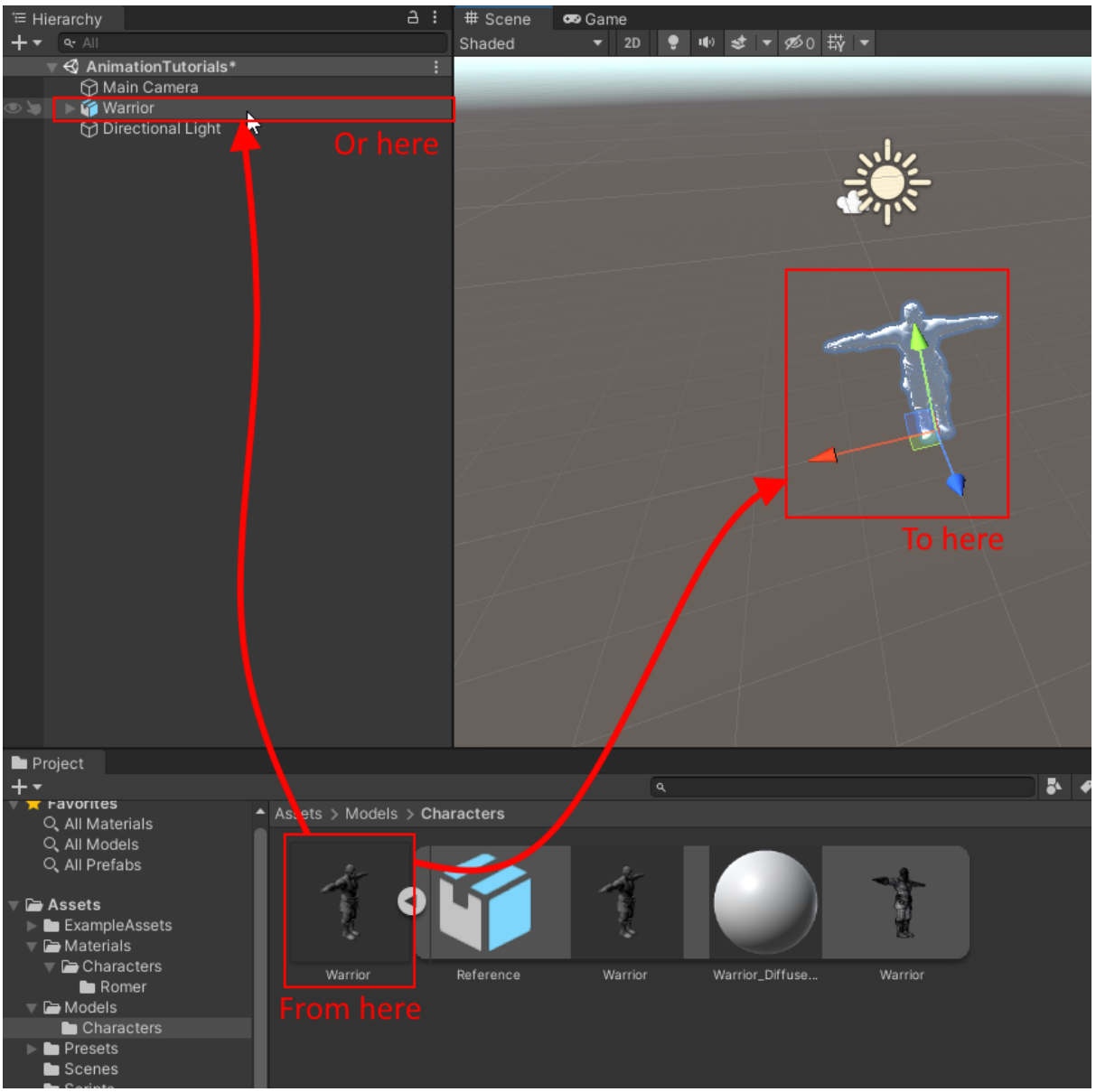
I'd similarly make the folders **Models -> Characters** and move the FBX-file for the character in there (no reason to give the model its own folder when there's only one Rømer model). Then I'd create a **Characters** folder inside of the existing **Materials** folder and put the material there. Complex models often have multiple materials, but in this case we'll be just fine with this folder structure.

It is also common for people to make an "Art" folder and put all of your art-related folders inside of that one, as it is a nice way to separate everything into art and logic, for example. I have not done this here as this is a very simple project and I am a bit lazy.

2.3 Create and save character as prefab:

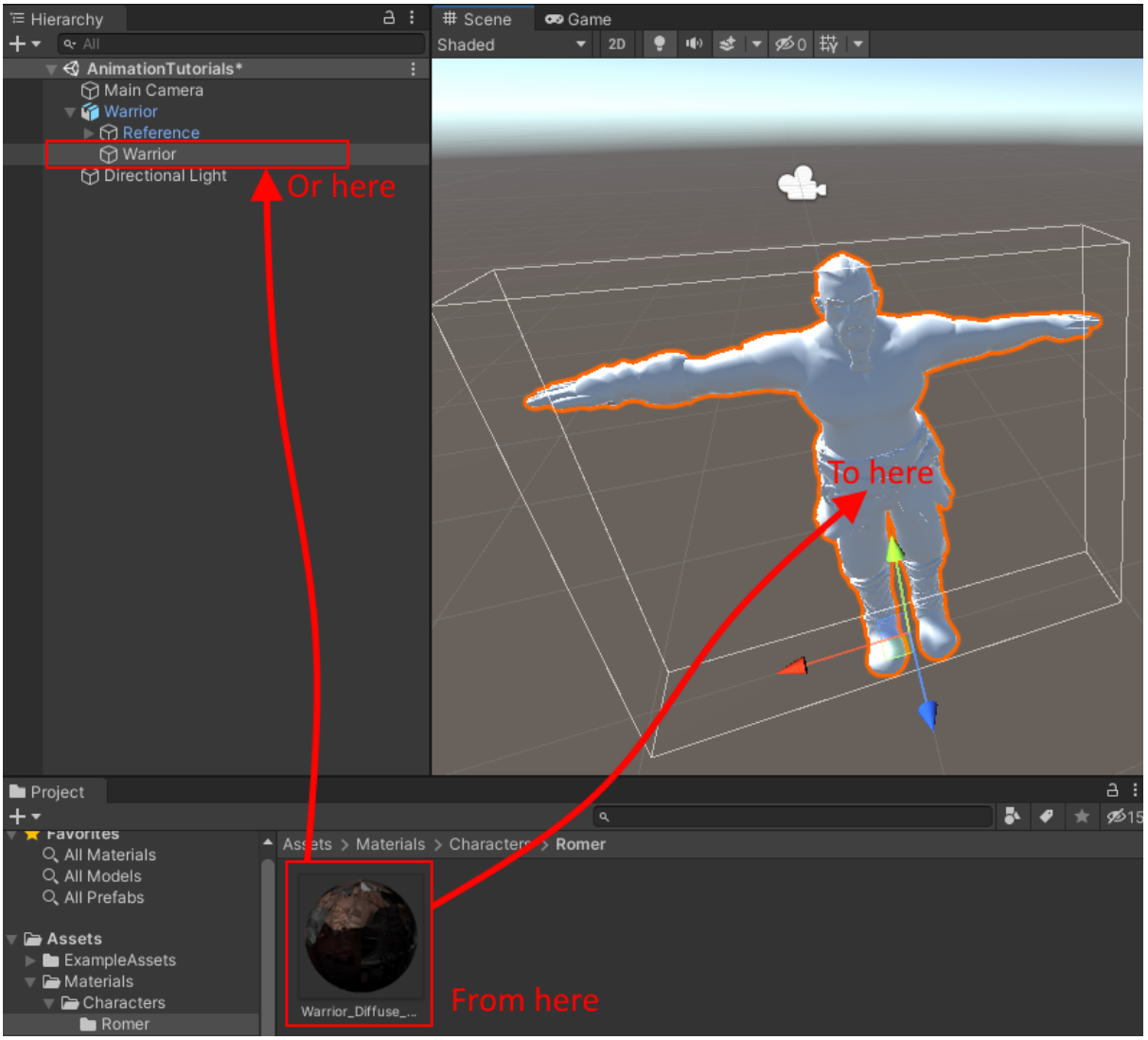
The last thing we need to do is to reassociate the material we made with the character model, then save the result as a prefab. Prefabs are really just blueprints for objects, in this case the character model with the new material, which we can easily reuse in our scenes without having to find and reapply the material every time we use the character.

Start by dragging the FBX into either the Scene window or Hierarchy window (contains a list of all objects in the scene):



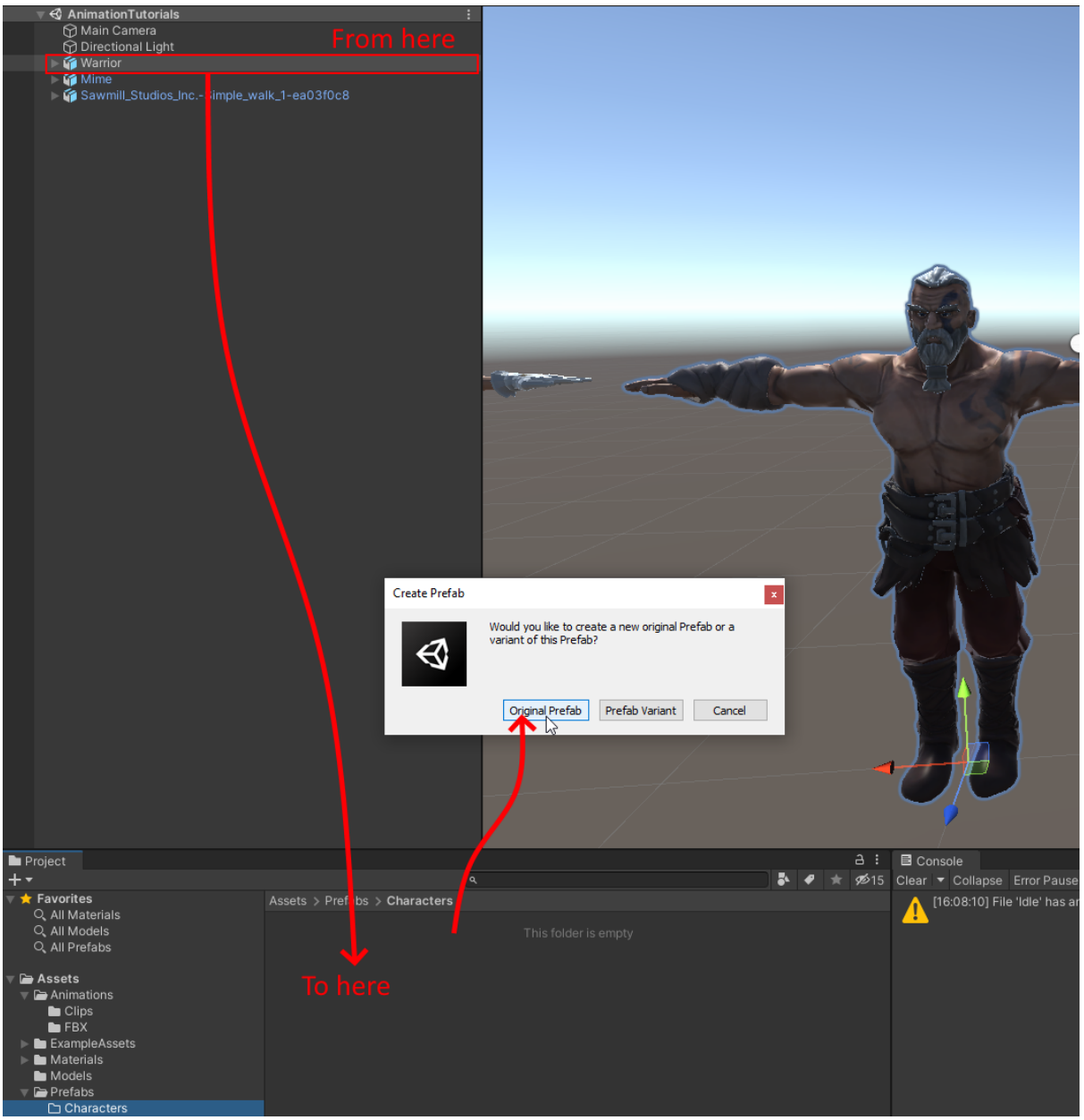
As you can see, our mighty viking warrior is also mighty pale, as the material in the fbx-file only looks in its immediate directory for the textures it needs. This sometimes happens when importing fbx files without dropping the textures into the project as well. Don't panic if it happens to you - that's why I am showing it here. To fix this we'll feed the mesh renderer the material we made ourselves. This can be done in two ways:

1. The easiest way to do this is to just drop the material onto the character in the scene, as Unity can identify the surface and find the mesh renderer automatically this way. I'll show how to do this below.
2. A more manual approach is to find the mesh renderer on the character, by selecting the character and assigning the texture via the Inspector instead. In most cases for characters like this, the mesh renderer is found just below the upper hierarchy object.



Alternatively, select the object with the mesh renderer and drop material into Inspector window

Next we'll be able to take the character object with the material and save that as a prefab. Just drag the Warrior object from the Hierarchy window into the Project browser, then Unity will ask if you want to save it as an **Original Prefab** or a **Prefab Variant.** For the purpose of this guide we'll choose **Original Prefab**:



If you wonder why there are two options, I'll refer to this guide for anyone curious about the difference between them, as prefab variants are a powerful tool.

2.4 Bonus practice:

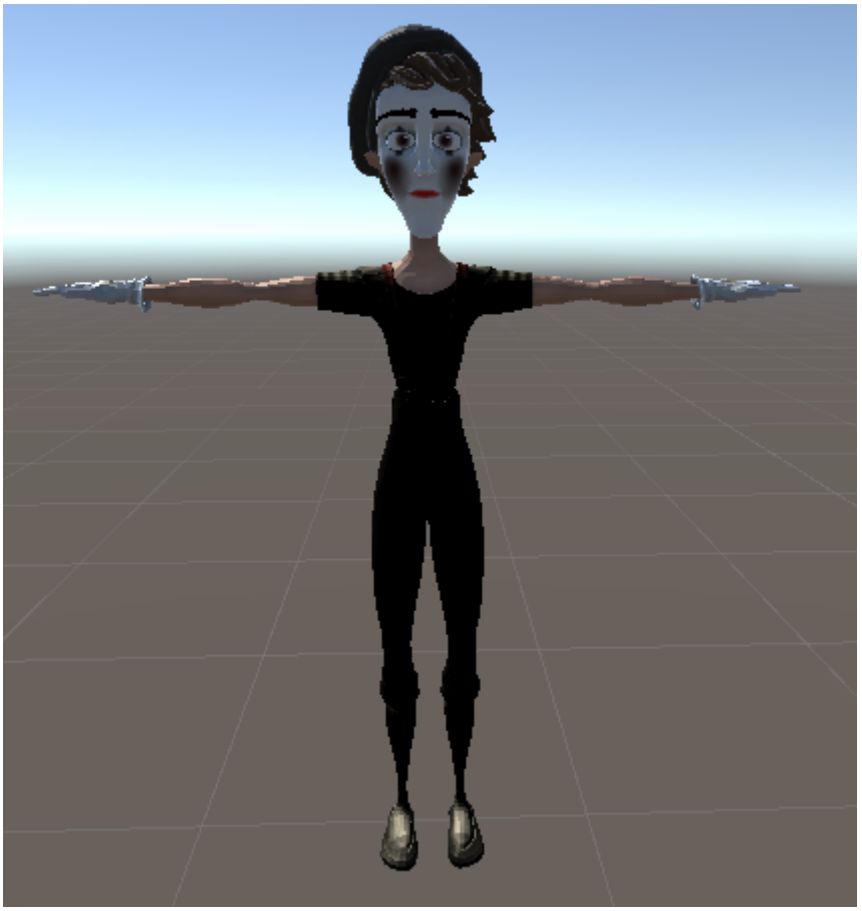
Now that you've mastered the art of splitting your fbx-files apart and putting the files in the right places, try importing the Bruno model and its textures in the same way. You'll notice that this time it has *three* materials instead of one. There's a few additional types of textures to be assigned as well:

- The **MaskMap** texture goes into the 'Mask Map' property
- The **Scattering** texture goes into the 'Occlusion Map'

And finally some tips:

- If named properly, textures should share names with the materials they go onto, so try searching for the name of the material you are editing, in the texture browser, to get all of the relevant textures to show.
- When applying materials to a character with multiple materials, the easiest thing to do is to drop them onto the character in the scene view, on the part of the character they are supposed to be applied to (eg. ‘eyes’ material on eyes, ‘hair’ material on the hair etc.).

The final result should look something like this:



And with that we have covered the basics!

2.5 Useful links (optional):

From Unity docs:

- General texture documentation in Unity
  - Texture import info
- Materials introduction

3.0 Retargeting:

Source: How to Animate Characters in Unity 3D | Animator Explained

Retargeting animation in Unity is inherently supported by the engine, using the Humanoid rigging system.

3.1 Process summary:

When importing animations into Unity, both the animation and the characters we wish to use it on, can be transformed into a “humanoid” avatar. This essentially standardizes the animation clip, so that it can easily be applied to all characters who also have their rigs converted to Unity’s humanoid avatar system. The process for the animation clip can be summarized as this:

1. Import animation fbx (eg. a mocap recording)
2. Convert animation fbx to humanoid avatar
3. Duplicate animation clip from animation fbx (to detach from its read-only state in the fbx)

When you have the animation clip ready, import the character you want to animate and do the following:

1. Convert character rig to humanoid avatar
2. Create and attach animation controller to character
3. Load animation clip into animation controller

With both the clip and the character using the humanoid avatar, the character should play the animation as the animation controller tells it to. We’ll get into multiple animations and blending between them in a later chapter.

The animation clips I’ll be using for this section are all free in the MotionLibrary. Their names are:

By Sawmill Studios:

- ‘Simple Walk 1’
- ‘Regular Run’

By SuperAlloy Interactive:

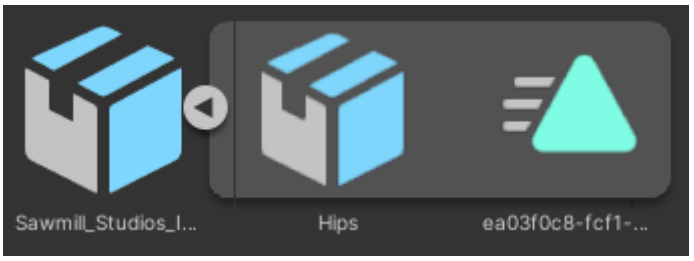
- ‘Idle Combat Relaxed’

If you are having trouble finding these, make sure that you are searching by ‘price ascending’ as that will list the free assets first.

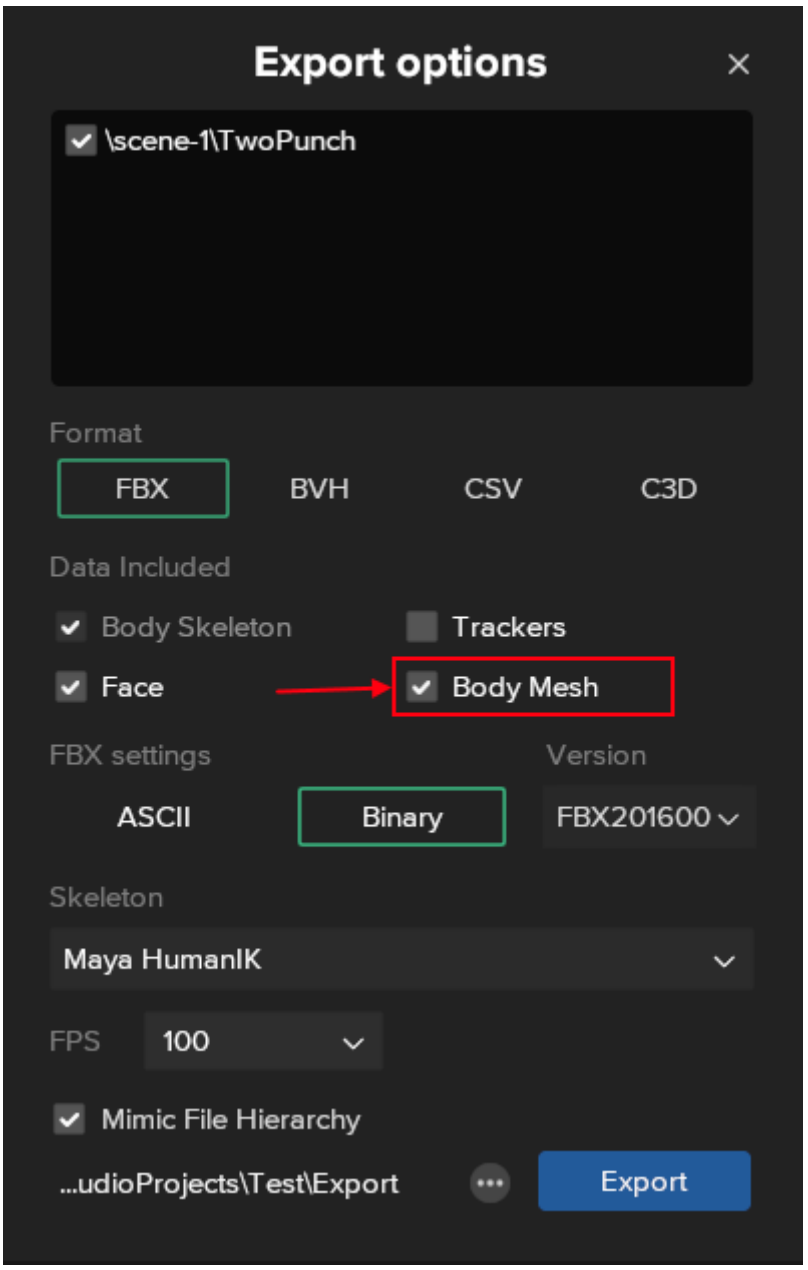
3.2 Import fbx with animation:

You’ve already done your mocap in Rokoko Studio and the potential animation cleanup cleanup at this stage (in Blender or Maya or whatever you prefer). To import the resulting animation, just drop the animation fbx into Unity. It should show up in the project window looking like this:

If you expand the details by clicking the arrow, you'll see that there is nothing here except the skeleton and the animation clip describing how the skeleton should move:



That means if you drop this fbx into the Unity scene, you won't actually see anything because there is no mesh associated with this file, only invisible bones and their transforms. If you do want to see how this looks on the Newton character in Unity, you can export the animations with the Newton mesh like this:

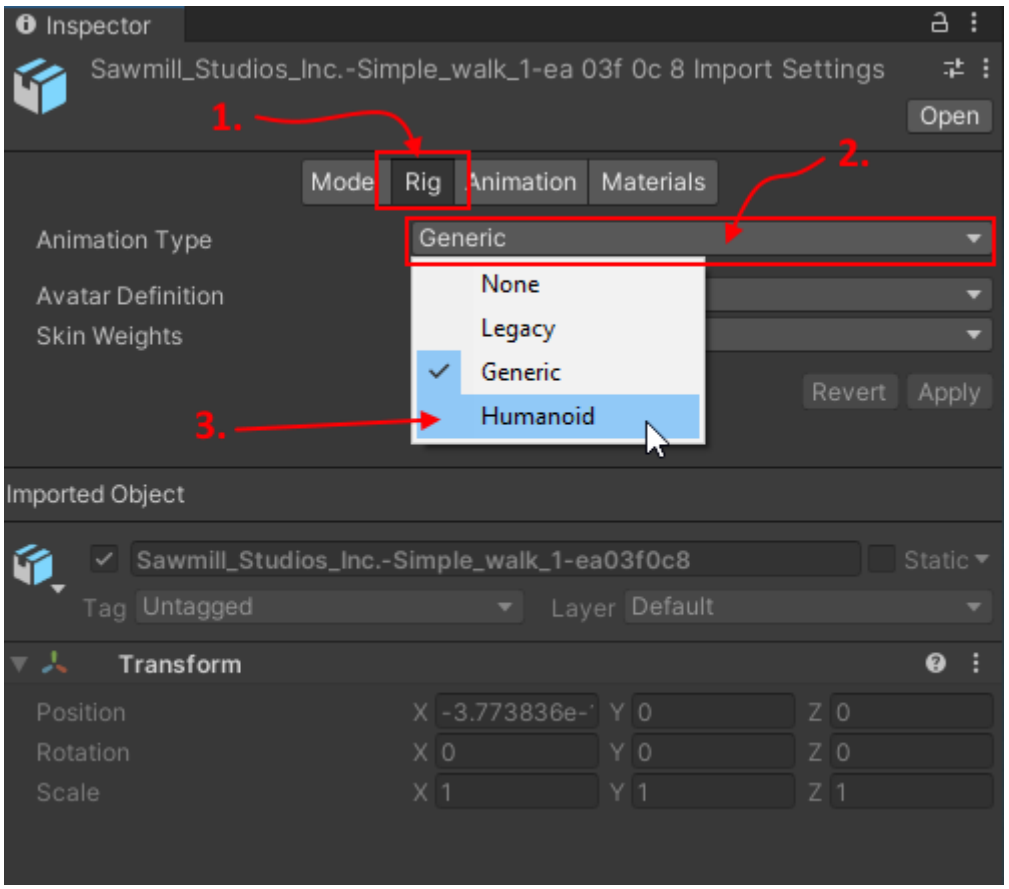


I also recommend using the **Maya HumanIK** skeleton - it is just a naming convention, it works perfectly for Unity as well.

3.3 Converting animation to humanoid:

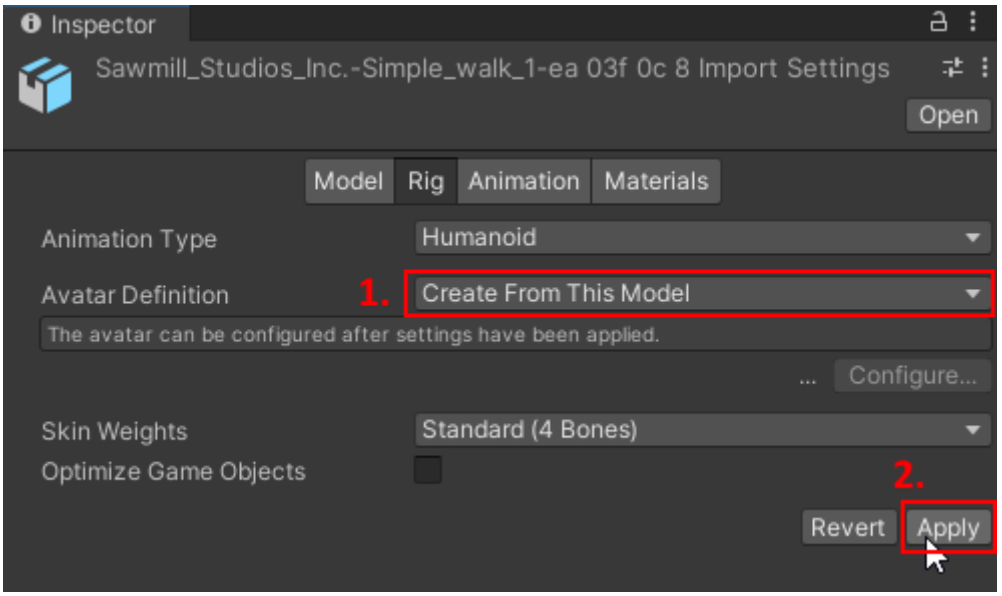
Now for a great magic trick, Unity lets us use the fbx containing the animation clip, to create a version of the animation that works with all humanoid characters. To do this, click the fbx file to view it in the inspector then:

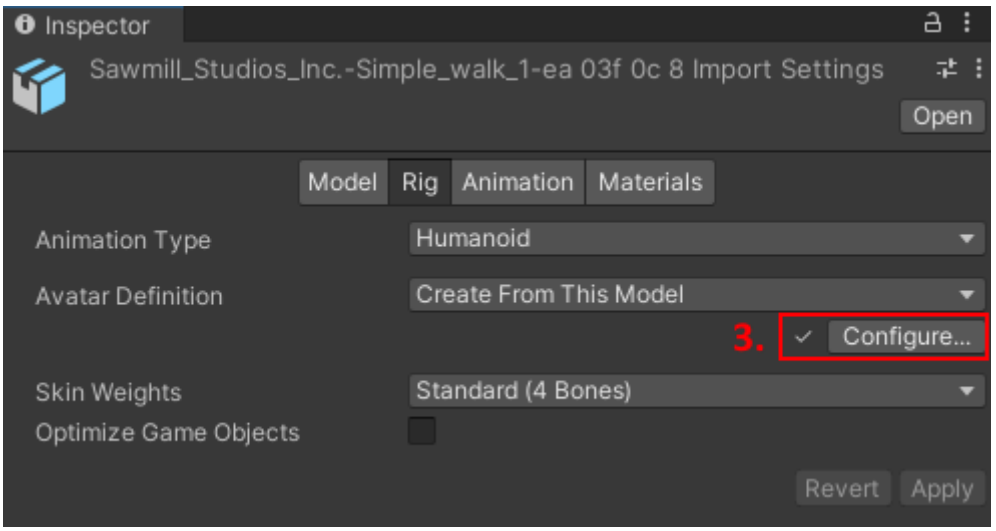
1. Select the ‘Rig’ tab
2. Click the ‘Animation Type’ drop-down
3. Change it to ‘**Humanoid**’



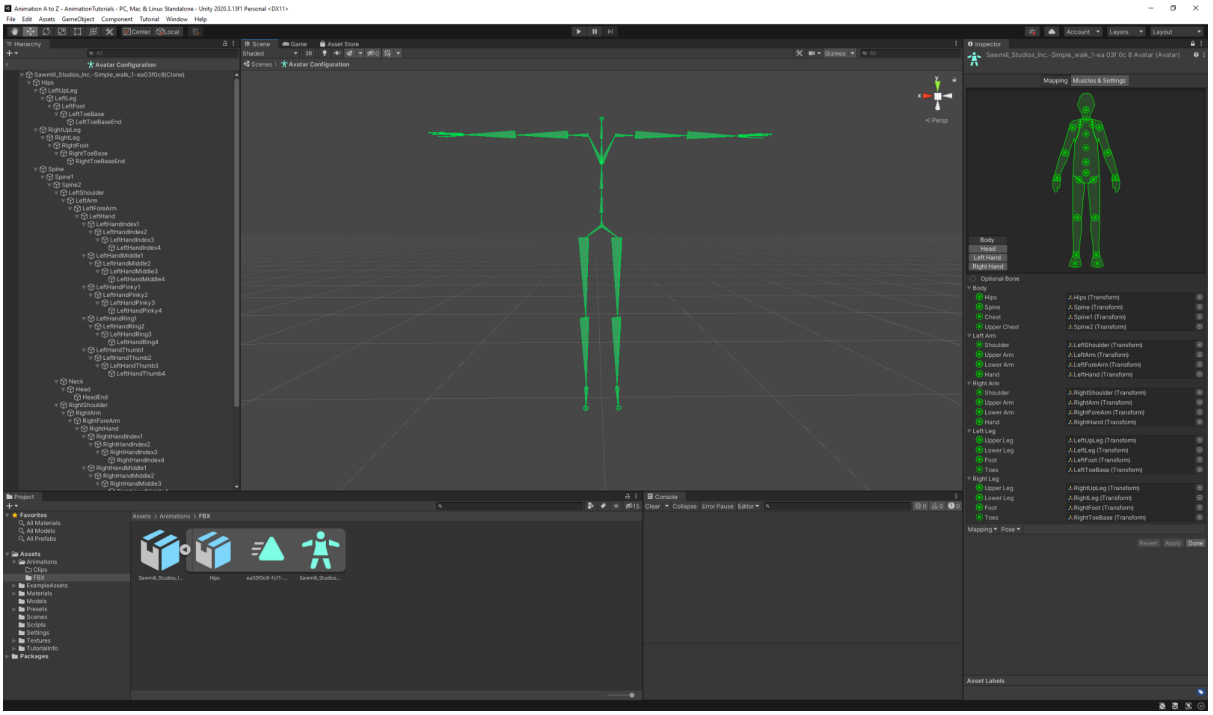
This will change the view a little bit as you will now have to confirm how the avatar should be defined:

1. Set Avatar Definition as '**Create From This Model**'
2. Hit Apply.
3. The previously greyed-out 'Configure' button should now be clickable:





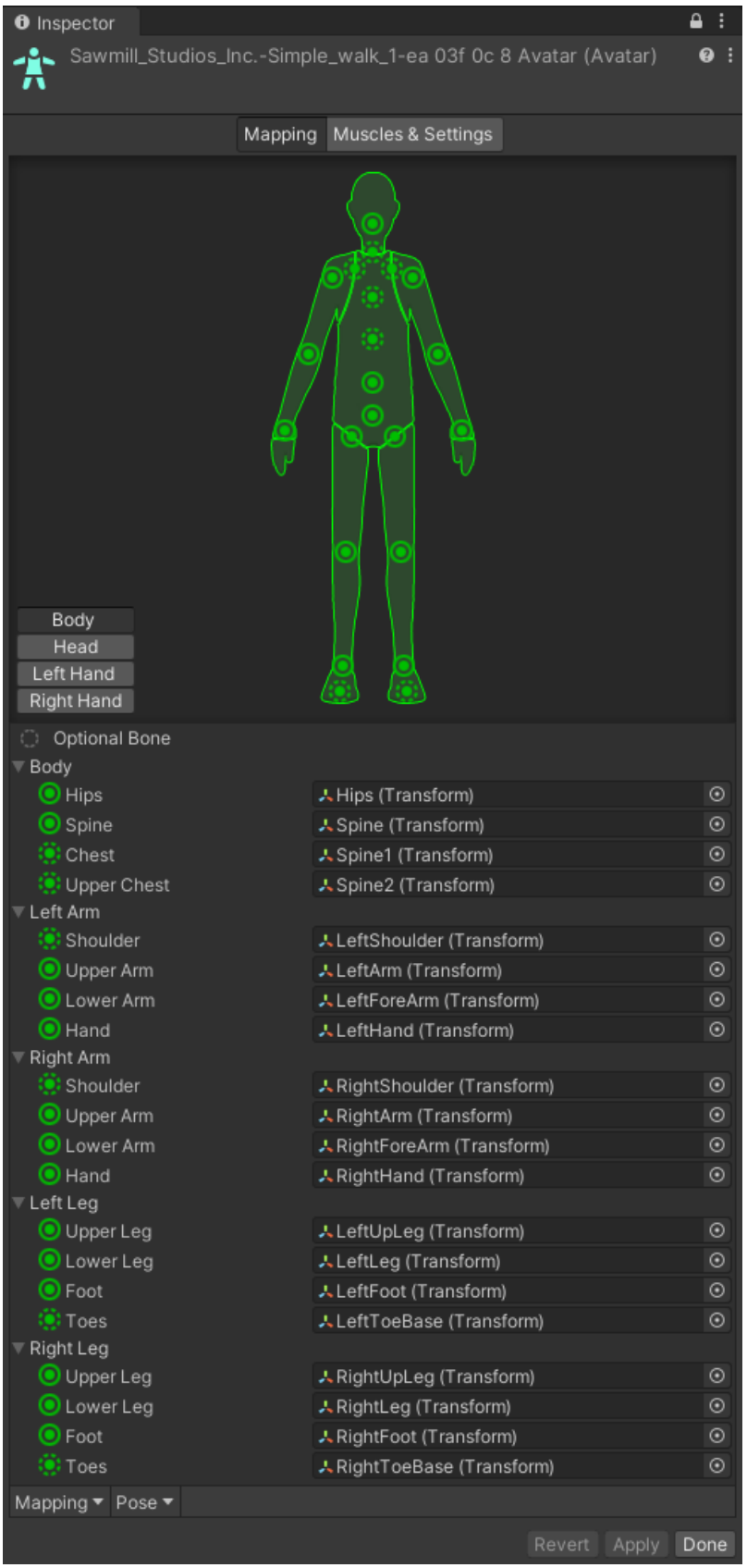
This will give you a window looking like this (depending on your layout of windows in Unity, of course):



To give an overview of what's happening in this tiny picture:

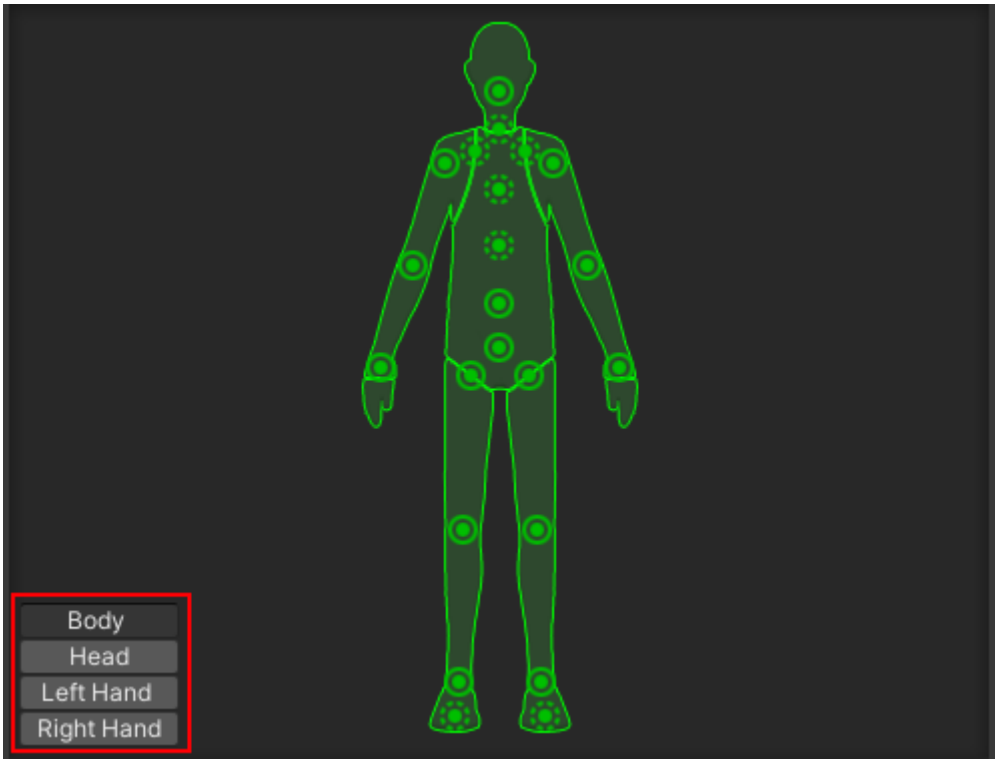
- **On the left side in the Hierarchy view**, we now see all of the bones from the skeleton in the FBX-file.
- **On the right side in the Inspector view** we see the simplified humanoid rig and the names of the bones that were chosen when remapping.
- **In the middle in the scene view**, there's a visual representation of the skeleton of the humanoid rig.
- **In the bottom left in the project view**, notice how there is now a little character icon in the FBX-file. This is how Unity shows that there is now a new rig associated with it (it is not actually *in* the FBX-file, since that remains read-only, but Unity shows it like it is, because it is easier to understand the relationship between them this way).

If we zoom in on the Inspector view, we can check if the bones from the original skeleton have been mapped correctly to the humanoid rig:

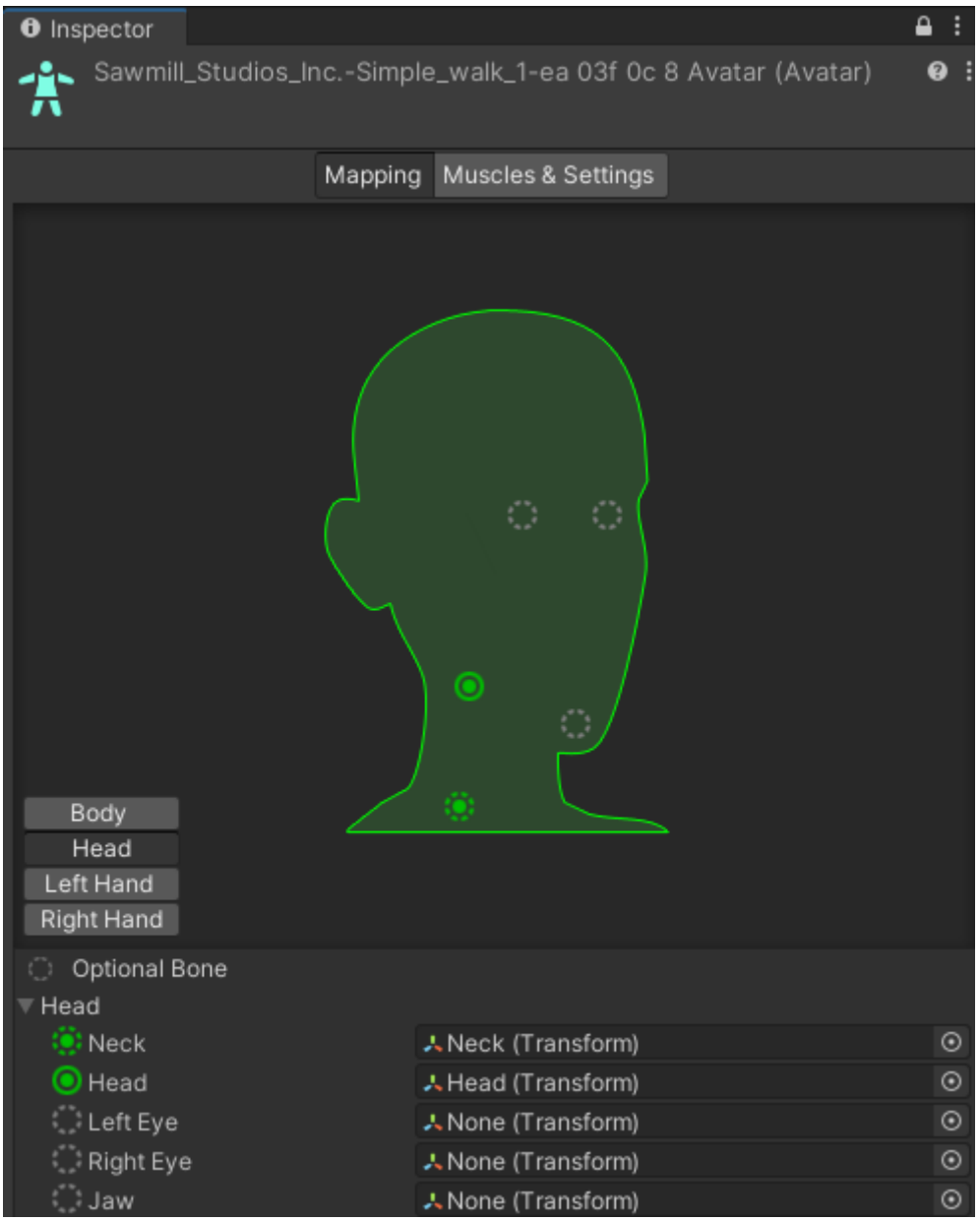


The names of the bones that were chosen for each part of the body, are shown to the right of the body part names. By clicking the green dots on either the character at the top, or to the right of the body part names, the bone chosen for that body part will be highlighted in both the scene view and in the hierarchy view. This is a great way to verify if the correct bone has been chosen.

Also notice how there are four buttons next to the green character:

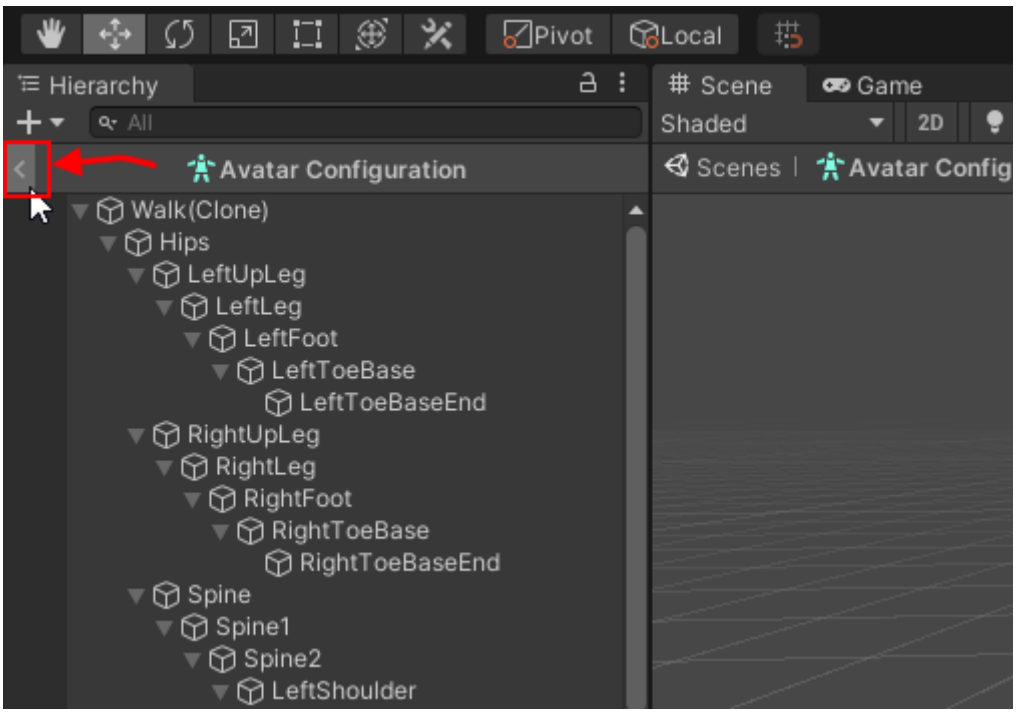


Clicking through these will make it possible to cycle through the layouts for those parts of the character as well, to see if all body parts have been named. In some cases, the original skeleton may not have all of the bones that are possible to map to the humanoid rig - but that's okay. As long as you are able to map to the body parts that have a solid green circle, then the parts with the dotted circles can be omitted, as they are not critical for baseline functionality of the rig to be working. For example, here is what the head of the humanoid rig looks like, when retargeting with the Simple Walk animation mentioned above:



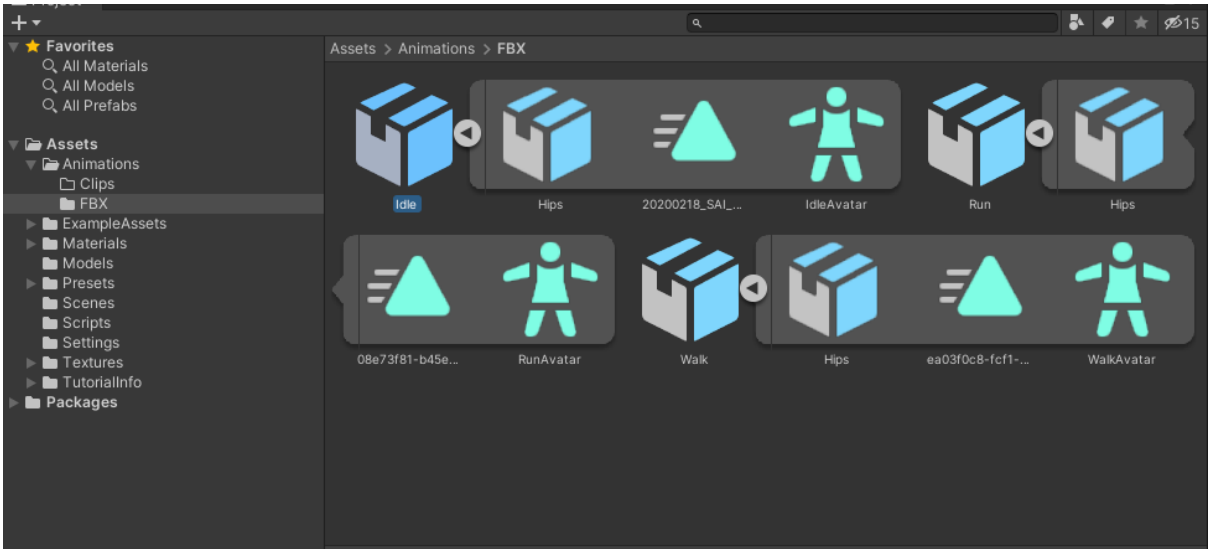
Several of the bones are not present in the skeleton for this animation, but seeing as they are all shown with dotted circles, it is okay that we omit them. And that's all it takes to get the animation clip ready to be copied from the FBX and used on other humanoid rigs. **Please note if you are using the SuperAlloy Interactive file**: *The rig is a little bit different from what we commonly use when exporting from Rokoko Studio - but it still works perfectly fine with this method, so don't worry if stuff like the hands are greyed out, it just means the clip doesn't have hand animation in it.*

When you are done with this and want to return to the scene view, look at the top left of the Hierarchy view, where all of the bones are displayed, there's a little '**<**' arrow you can click to exit the Avatar Configuration view:



3.4 Renaming and sorting your animation clips:

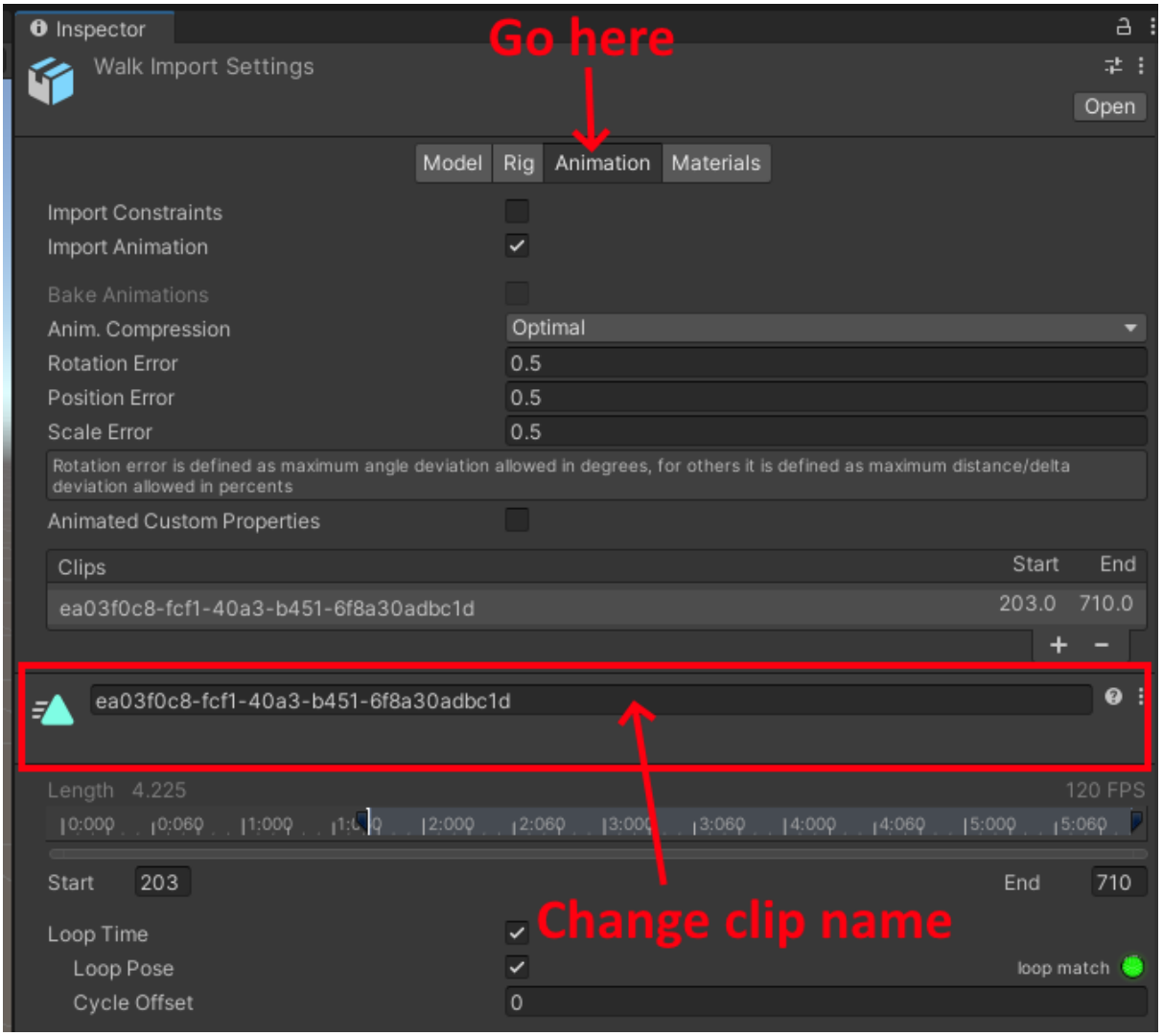
Here's a look at the project folder where I put the animation clips from Motion Library. I've set all three of them up with the humanoid rigs and renamed them from their original names to 'Walk', 'Jog' and 'Run' which is infinitely easier to read at a glance:



Note from this picture:

1. The Humanoid rigs (avatars) are inheriting the names from the FBX, so it is important to keep them simple.
2. The actual animation clips inside the FBX files are sometimes called some completely random nonsense. To address this, you will need to **select the FBX file**

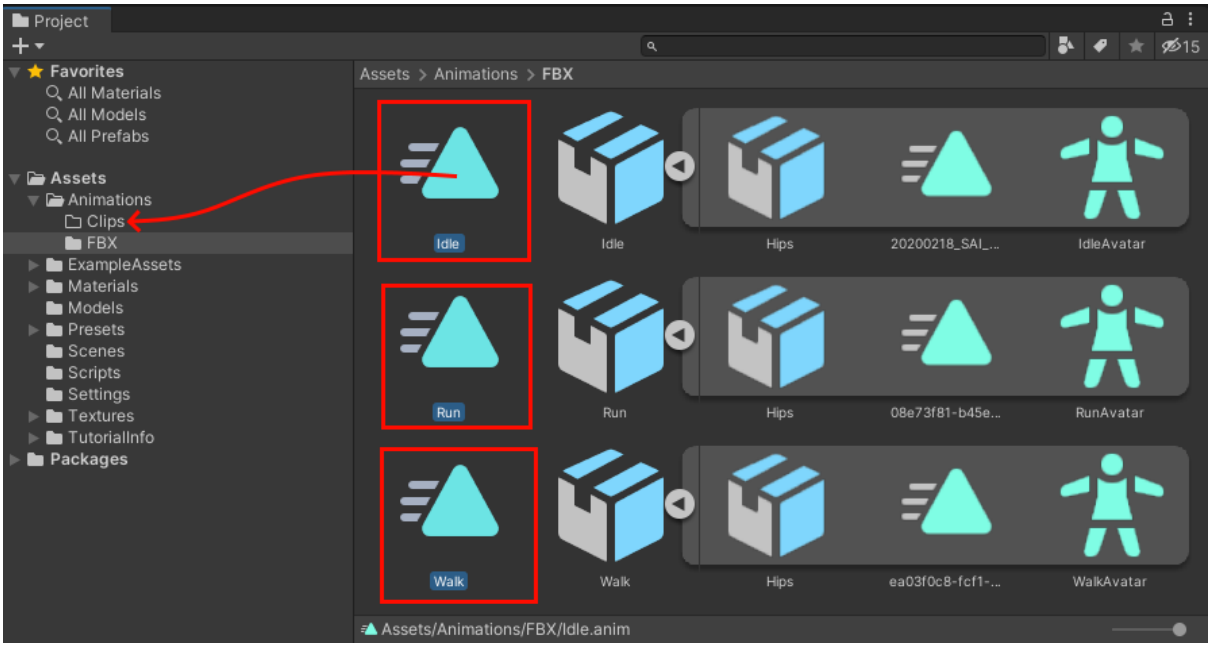
itself (rather than the clip directly), **go to the Animation tab** and change the name there instead:



Here you can change the name to whatever you want - in this case ‘Walk’ is the most appropriate. These are the names that show up when searching for animation clips, so make sure to use something that describes the animation well.

3.4.1 (OPTIONAL) Duplicating the animation clip:

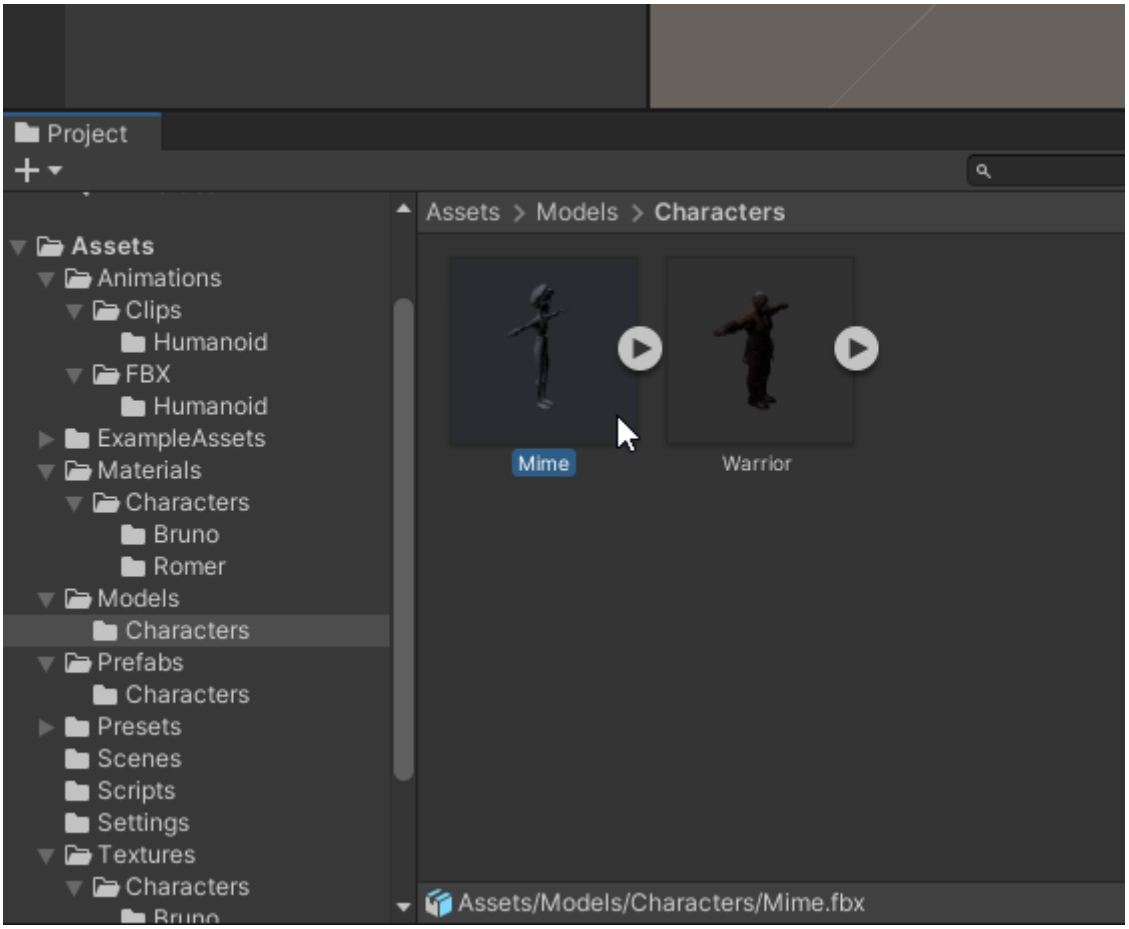
This is NOT standard practice, but there are some cases where you might want to edit the keyframes directly within Unity. This is not possible when looking at the clips while they are contained in the FBX file, but you can duplicate the animation clip to overcome this, at the expense of not having access to some of the tools that Unity has when working with the FBX directly. *If you do not know if you will need this or not, you probably do not need it, but it is good to know that it can be done.*
To duplicate a clip, make sure you are looking at the expanded FBX file in the *Project* window, select the clip and press CTRL+D to create a duplicate outside of the FBX file. Then make sure you store them somewhere that makes sense, so you do not have to deal with anim-files and FBX-files cluttering up the same directory:



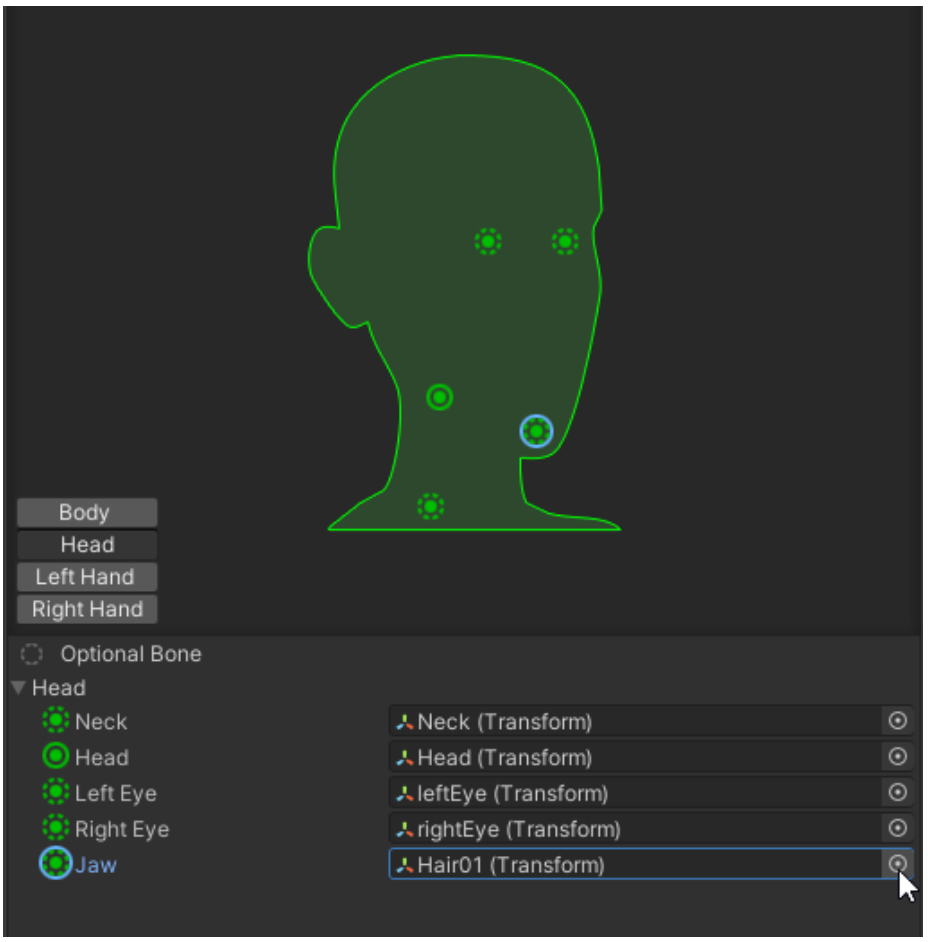
3.5 Convert character rig to humanoid avatar

With the animations named, we need to set up our characters with humanoid rigs as well. The procedure for this is actually identical to when we configured the humanoid avatars on the animation clips, so I am not going to go too much into detail with this, just a summary:

1. Select the original FBX from when you imported the model (like the untextured mime):

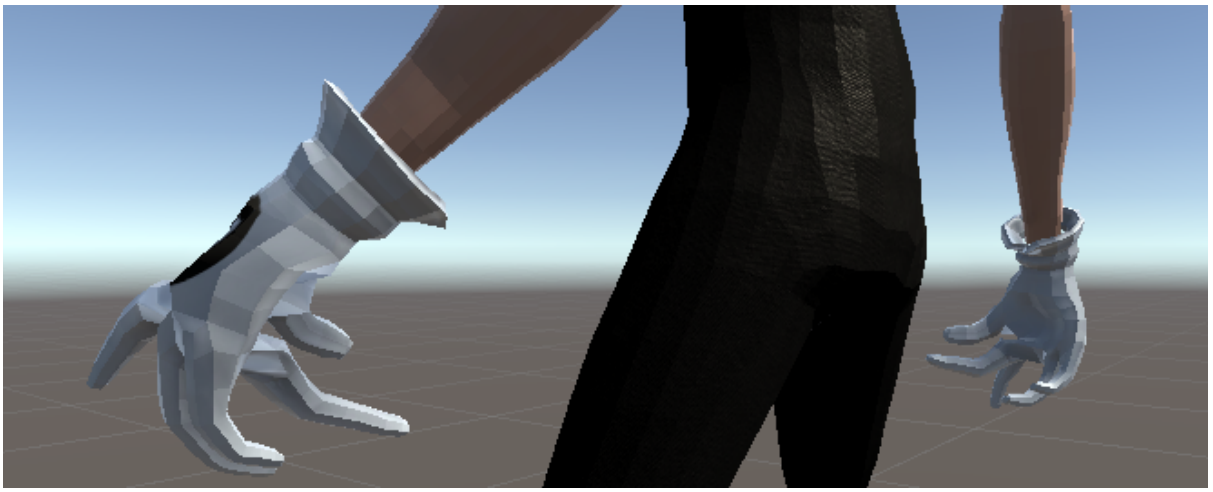


2. Go into the ‘Rig’ tab in the inspector
    - Set ‘Animation Type’ to ‘Humanoid’
    - Make sure the ‘Avatar Definition’ is ‘Create From This Model’
    - Hit ‘Apply’
    - Go to ‘Configure’ to double-check that the bones are mapped correctly (for example, the Mime model will likely have one of the ‘Hair’ bones mapped for the neck bone, that should be unassigned:

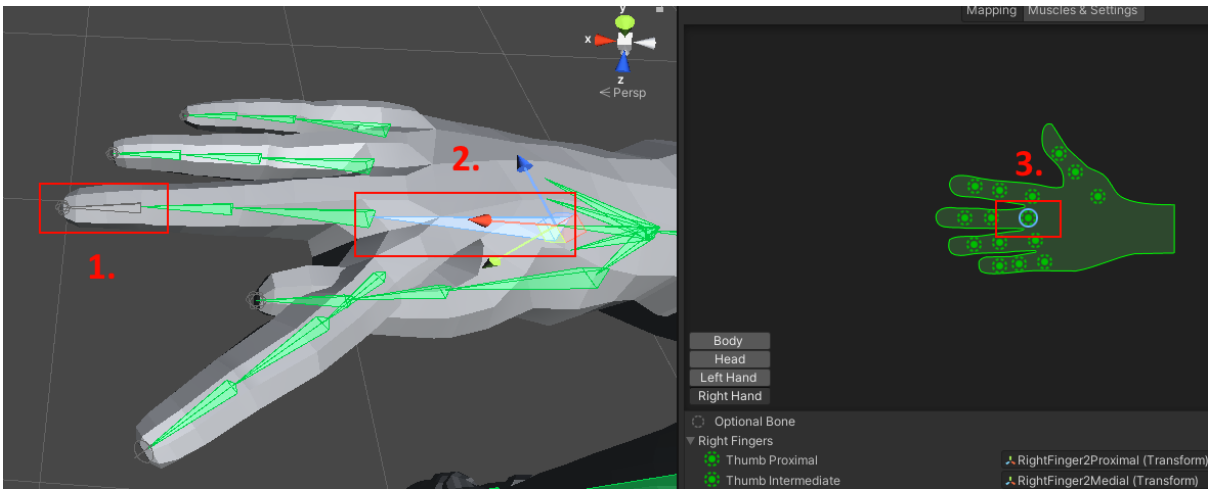


Click the grey circle on the right -> Select 'None' to deselect bone

Remember to go through the bones like this, if you ever have any problems with your characters posing weirdly. For example, the Mime character did not have its finger bones mapped correctly by the automatic mapping process, so if you play any animations that affect the hands, you will find that it is looking more like a bowl of spaghetti than actual fingers:



The easiest way to double-check your mappings, is to open up the Humanoid rig configuration tool again, focus on the hand and click on the bone directly in the preview, to see if it matches the location of the visual aid in the Mapping overview:

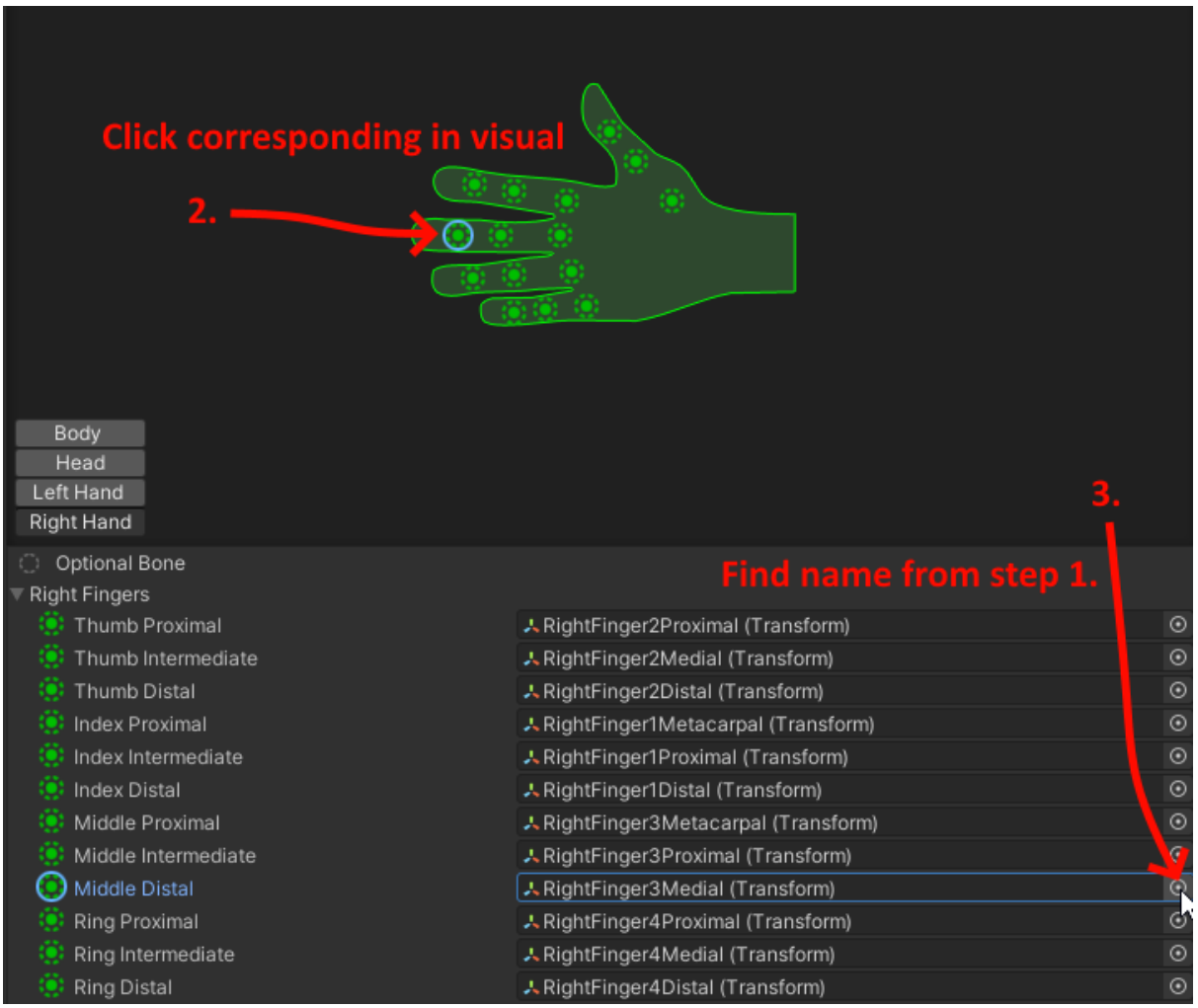


If we for a moment ignore that the T-pose is weird, I have outlined three things which indicate that something is wrong:

1. The tip of the middle finger is not mapped, even though all the other fingers are (mapped bones are highlighted in green).
2. The middle bone is mapped, even though the humanoid rig (on the right) does not have a marker for this bone in the visual.
3. Selecting the middle bone confirms that - by looking at the visual - the automatic mapping has mistaken it for the first digit of the middle finger (hence the missing tip from point 1.).

To fix this, click on the bone that you wish to map, to find its name. Then go into the bone map and find it, so that it gets mapped to the right place:

To put it shortly - if in doubt; click through the bones on the visual, and check that each of them are mapped correctly on the actual skeleton.

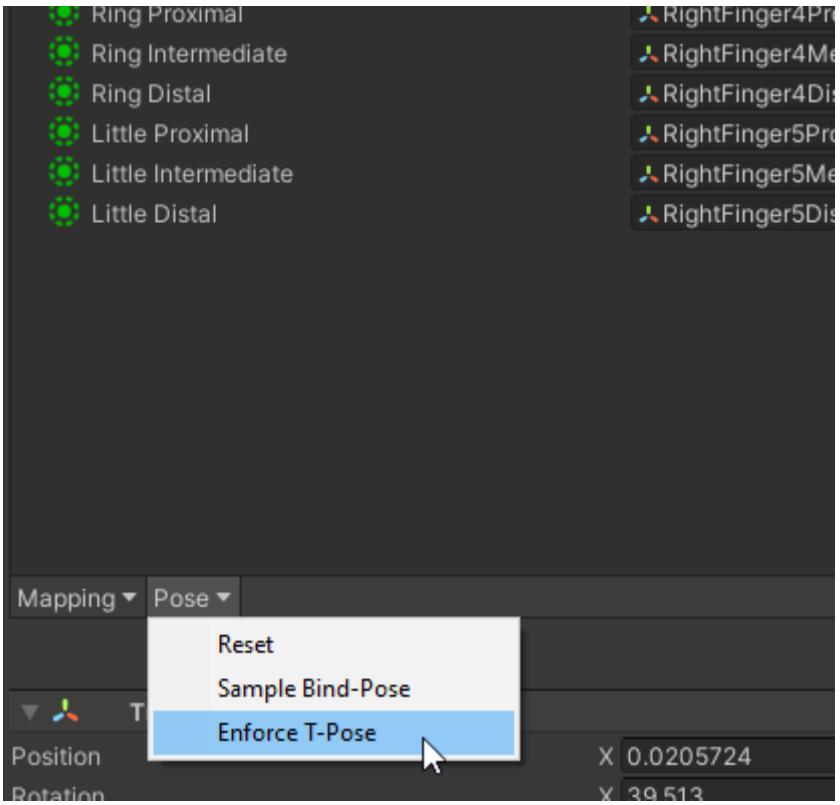


3.5.1 Bonus practice (hand surgery):

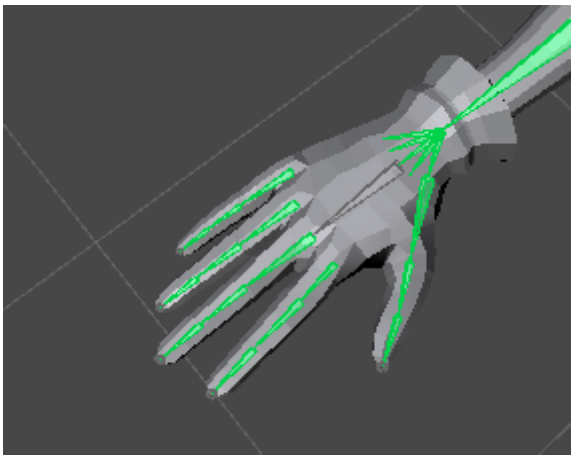
On the Mime model, if you click through the bones on the list or on the hand, you will notice that there are several problems with the mapping - likely from the T-pose and bone layout being a bit different. As a practice, I recommend that you try to click around and try to identify where these problems have occurred (though on the hands only) to familiarize yourself with the process.

*Please note; you **will** see errors on the bone list when you start remapping, as some of the bones will temporarily have the same names when you swap them around. Just keep remapping and the errors will disappear when all of the bones are mapped correctly.*

After you are done, it will likely complain that the character is not in T-pose. If you look below the bone list, you will notice that there is a button that says ‘Pose’. If you click this, there is an option to enforce the T-pose, which will reset the pose of the character to the T-pose:



And voilá, hand surgery complete:



4.0 Animation basics:

Now that we have characters and animations available, it is finally time to go through the animation systems which Unity provides us with! There are several tools and logic flows that we must familiarise ourselves with, to really get the most out of everything. The short summary of the tools that we'll explore are as follows:

1. **The Animation Controller (and Animator component)**

Dictates how and when what clips are played on a character. These are coupled with an 'Animator' component that sits on the character, which is what actually animates the character, as dictated by the Animation Controller.

2. **The Animation Timeline**

This is where all the keyframes for the bones are listed. It is very similar to the same feature in other software such as Blender, which it is common to do cleanup and edits in, instead of doing this in Unity, if direct changes to keyframes need to be made.

3. **The Animation Rig**

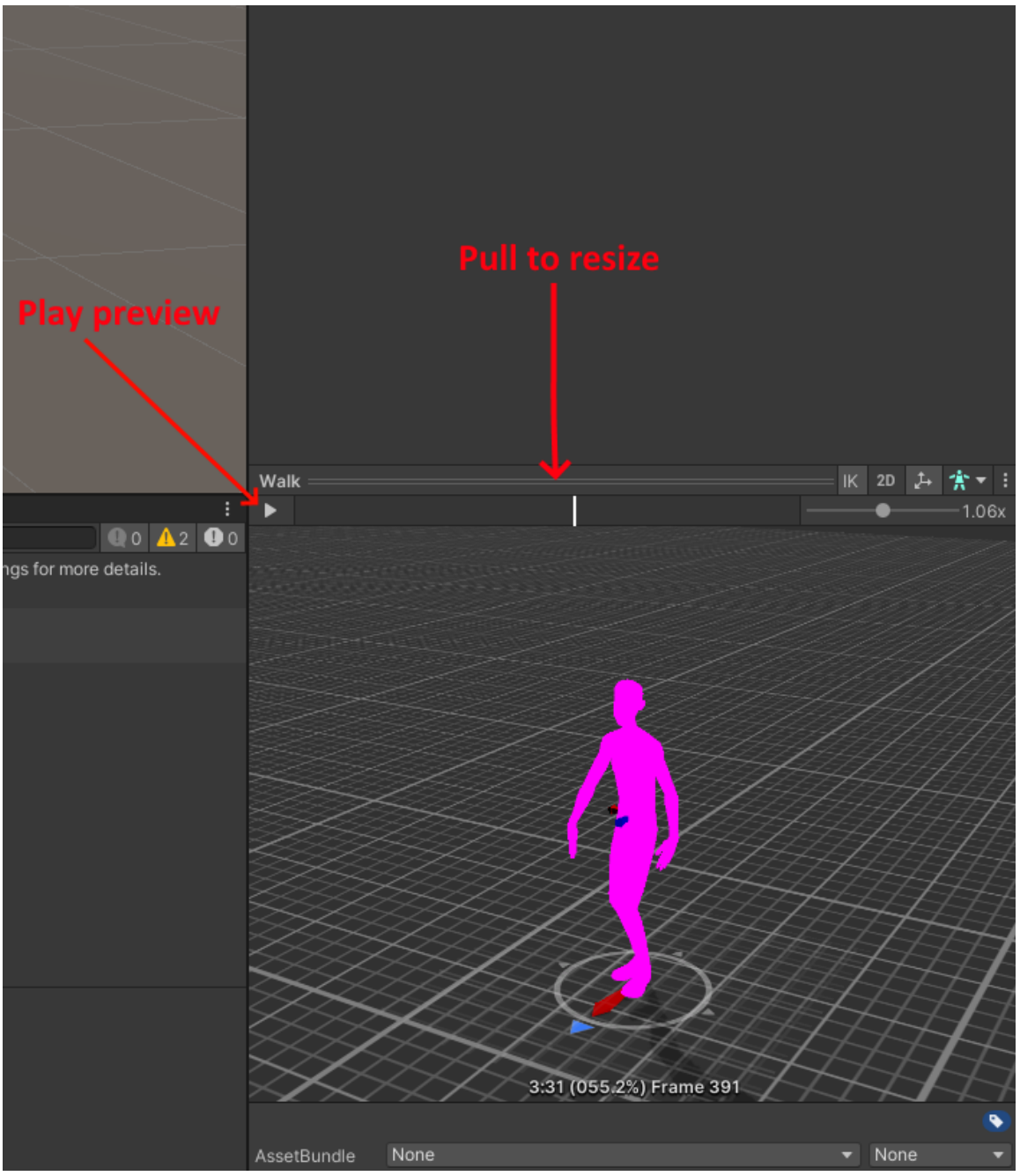
This is a feature that lets us set up constraints and override parts of the existing animation, which is amazing for dynamic interactions in a live environment, such as having to react to input from a player in a game.

4.1 Testing and modifying animation clips

Before we get too deeply into the logic of the animation systems, we first need to make sure that our animations work as intended. This is especially important for looping animations that were done by mocap and not by hand, if you have not already edited this to loop perfectly elsewhere like in Blender.

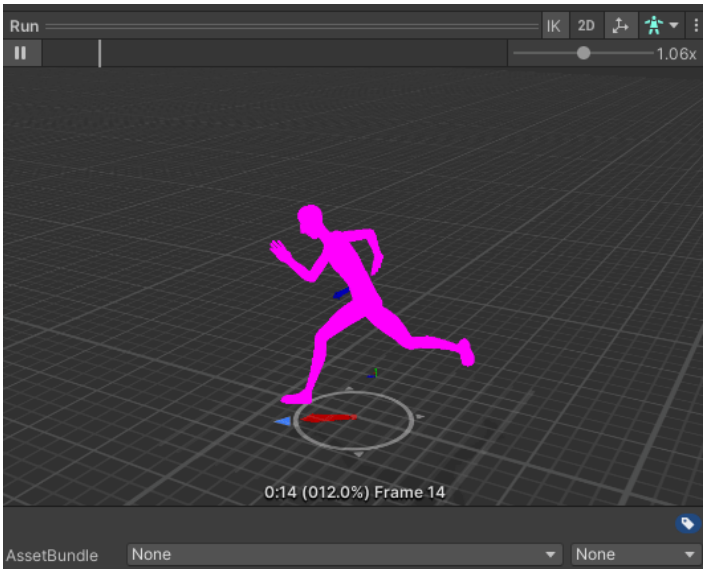
Unity actually has some awesome tools for modifying your animations without messing with the keyframes, to generally getting some really good results, so long as your mocap has a similar pose at least two places in your animations.

Let's start by looking at the run animation, by selecting the FBX that contains this animation, and going to the Animation tab to preview it in the inspector. It will show the animation view at the bottom of the inspector window, where you can press the play button to preview it:



If the preview window is too small, you can drag the bar that you see above the preview to make it bigger. If you can't see it at all, you may have clicked the bar and it has moved to the bottom of the inspector view - just click it again to restore the preview.

Here's what the run animation I've chosen looks like:

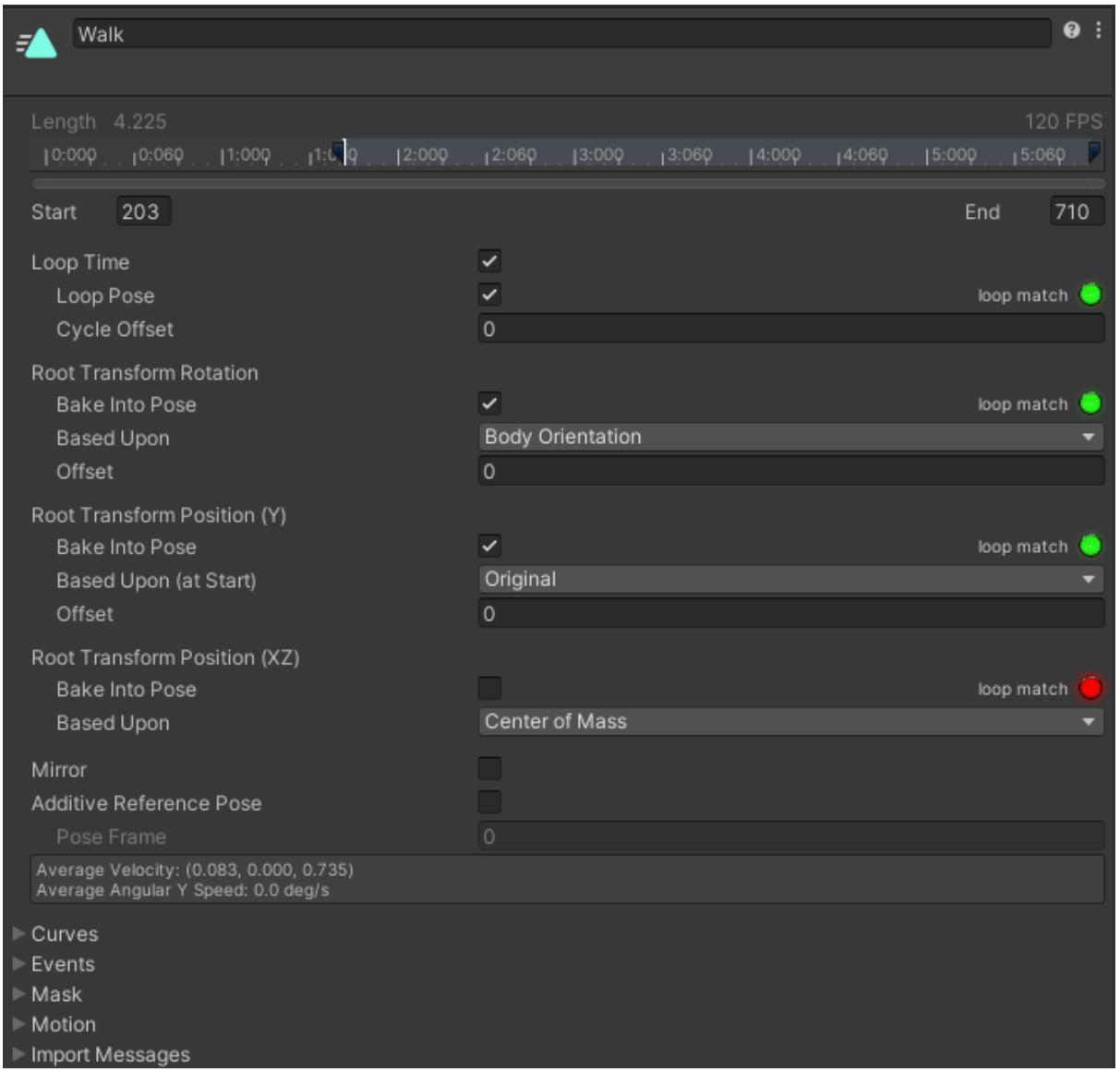


([PLAY GIF])

There's a few problems with this aside from the fact that it changes direction when it loops. Let's check out the settings that are available in the inspector window, to see what we can do about that:

4.1.1 The settings:

If we scroll past the initial settings (which are mostly related to import), we are greeted with a view that looks like this:



Your view might look a little bit different, but the options are the same. Here's a short summary before we get into using them:

Looping options:

- **The Length Slider** is the bar right below the section where we previously changed the name of the animation clip. Note how there is a value for **Start** and **End** with a number - this represents the start and end frames of the animation, which you can type in manually as well.

- **Loop Time** lets us loop the animation. While it does look like it is looping in the preview, it actually does not do this when using the animation in practice.

- **Loop Pose** tries to match the pose of the first frame in the animation, towards the end of the animation. This is amazing for walk or run cycles where you need the looping to be seamless.

- **Cycle Offset** lets us determine what frame the animation starts at, but doesn't omit the frames before it (so it's no good for cutting away a T-pose at frame zero, for example, but great if you need to start a loop at a specific pose).

Root Transform Rotation:
Before I explain the root transform rotation, we need to understand what the root transform is. The simple answer is this: The 'root' is the one part of the skeleton that has a global coordinate set and rotation - aka coordinates relative to the world. For human skeletons, this is commonly the hip bone or similar, meaning that moving this bone will move the rest of the character as well.
All other bones have their coordinates and rotations set relative to the root bone, rather than world space, so that moving a character around by the root, will not mess up the animation.
If you want a deeper explanation, I recommend checking out Unity's official documentation.

With that out of the way, here are the options for root transform rotation and their explanations:

- **Bake into Pose** means that the root rotation is constant, disallowing the animation clip from modifying the rotation. You can of course still have rotations in your animation, but they will not rotate the actual character object.
- **Based Upon** lets you choose how the rotation is then determined:
    - **Body Orientation** uses the forward vector of the body as the direction.
    - **Original** will use the direction that was originally set in the animation clip, which can be useful if there is a scene setup where the direction has already been planned and implemented into the animation.
- **Offset** is for adding an offset to the direction that the character is moving. This is particularly relevant when using the *body orientation* option for direction, but the animation clip is the character strafing sideways rather than running forwards, for example.

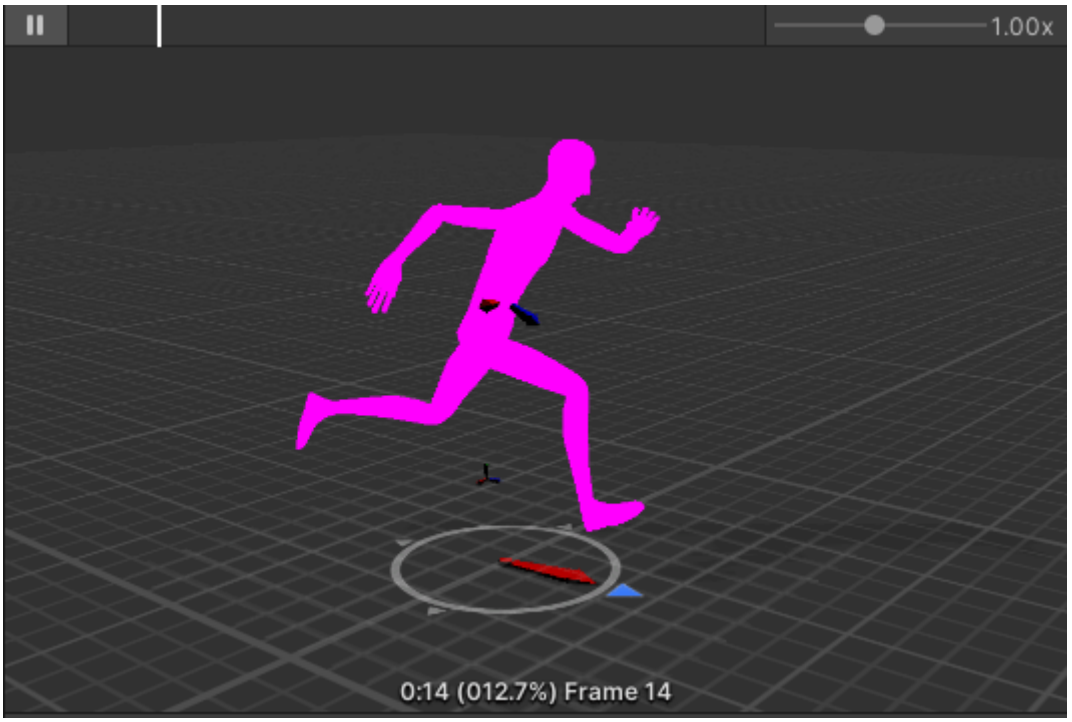
Root Transform Position (Y):
This setting lets us modify the elevation of the clip, in relation to the root transform. It applies an offset to the Y-axis, with a few presets that lets us modify how those are calculated:

- **Bake into Pose** is the same as with the rotation - we toggle whether we allow the animation clip to make any changes to the Y-coordinate of the root transform. You want this to be off for most of the time, except for animation clips that are supposed to lift the character, like when performing a jump.
- **Based Upon** lets us make changes to the offset, based on three different parameters:
    - **Original** uses the same height offset as the original clip.
    - **Center of Mass** uses the middle of the character, calculated from what I believe are the edge vertices, as this can shift throughout the clip.
    - **Feet** uses the contact point for the feet on the ground, to determine the height. This is useful for something like a run animation, but only if you are not baking the height into the pose.

Root Transform Position (XZ):
Similar to the previous feature, this one works on the horizontal plane. It can be useful for preventing issues like positional drifting over time, if a character is performing a repeated idle animation that doesn't return to the exact same position. I won't really go into details as it is mostly a repeat of the same feature as above.

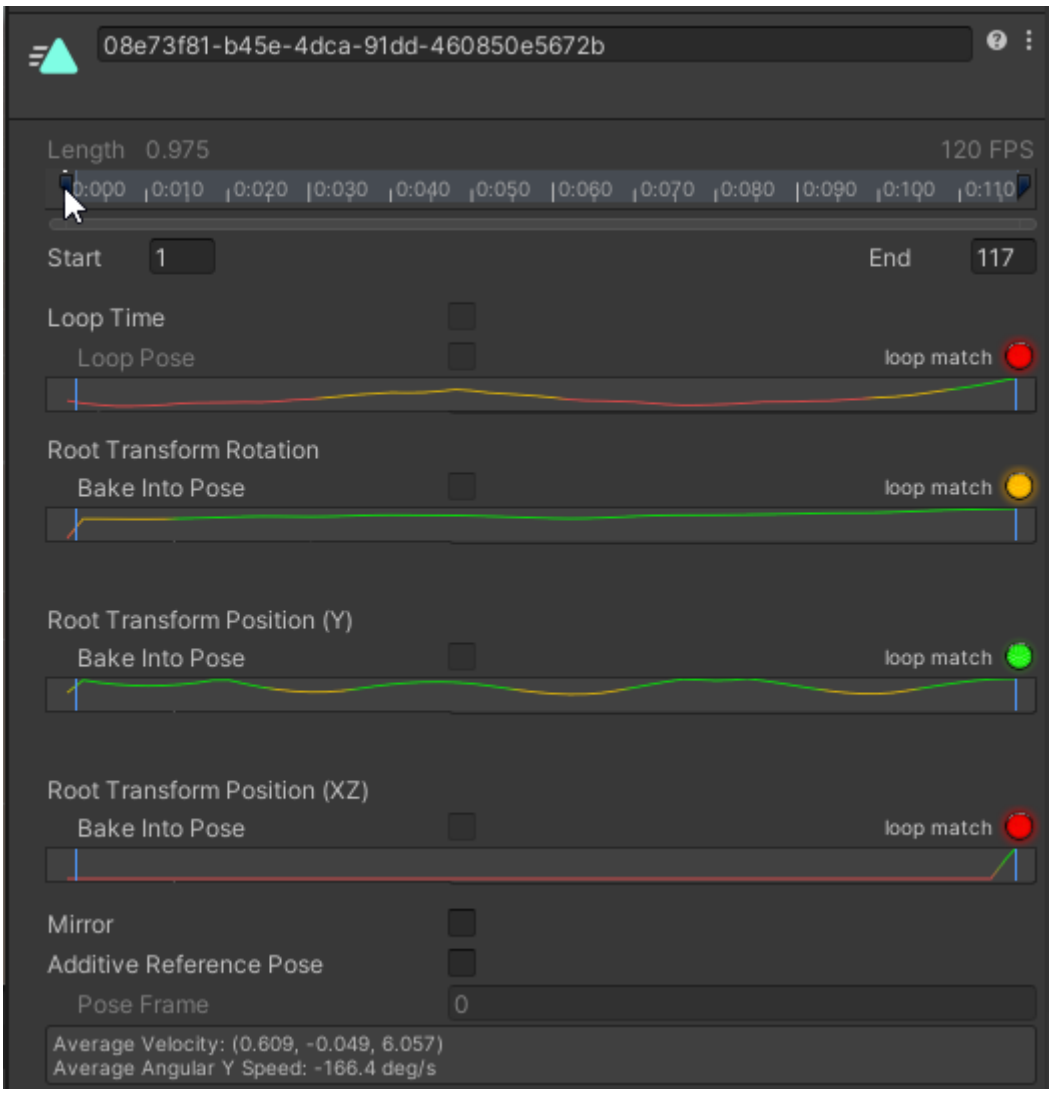
4.1.2 Making modifications

Now that we know the basic tools that we have available, let's get around to using them on our animation clips to get the best results.
The first thing that you'll notice when messing around with loops, *is that there's a T-pose in frame 0 (the first frame of the animation).* So if we turn on the options 'loop time' and 'loop pose', the animation will actually try to match the T-pose towards the end, which results in this contortionist masterpiece:



(PLAY GIF)

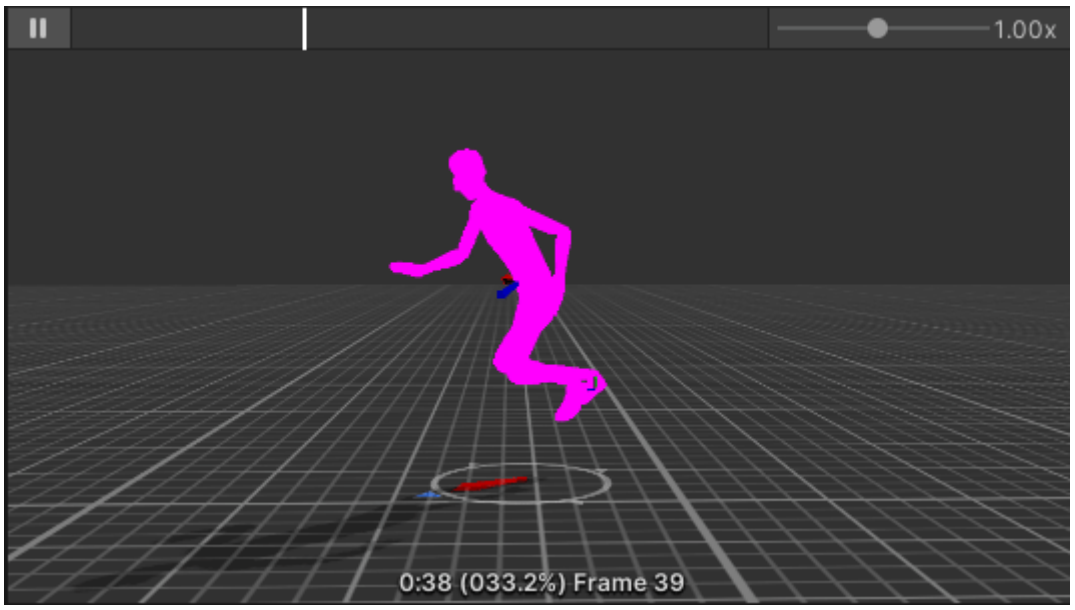
If we start messing around with the Length Slider for the animation, notice how the view changes below:



These coloured curves show how well each frame of the animation would line up with the end frame, for looping with any of the given settings. Red is a bad match, yellow is okay and green is good. The compatibility is measured by how close the two poses match, based on the transforms of the character. Let's take a few steps and see how the resulting animation changes:

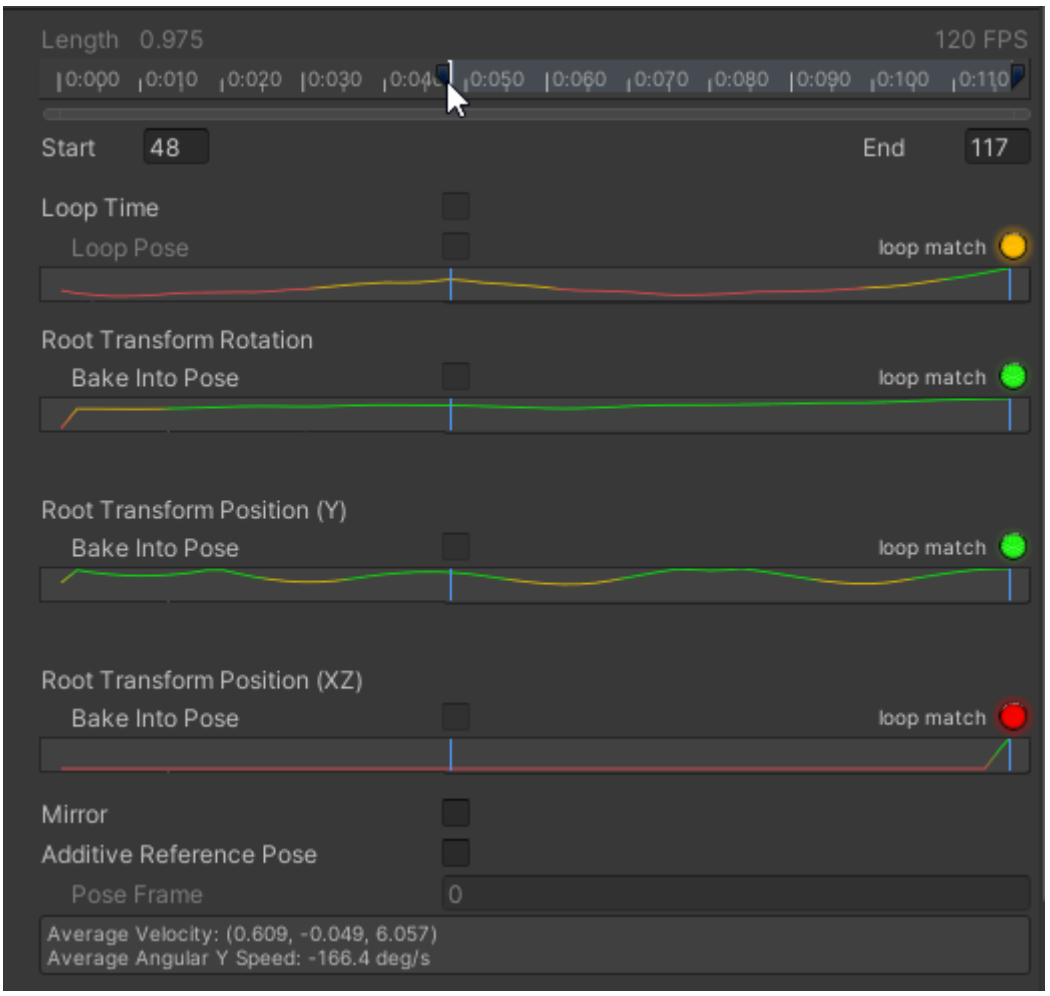
- Turn on 'Loop Time' and 'Loop Pose'
- Under Root Transform Rotation, toggle on 'Bake Into Pose' (stops the character from spinning)
- Right under the Length slider, set the value of 'Start' to 1. This skips frame 0, which is where the T-pose is.

The result is now this:

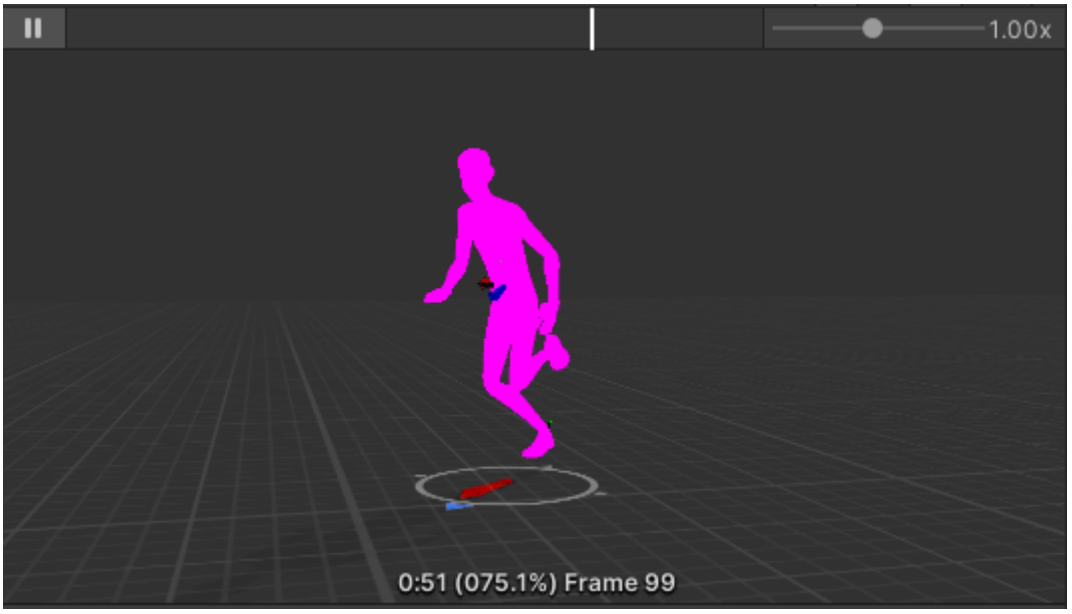


(PLAY GIF)

Still terrible, but slightly less so! The reason for this is as the curves in the previous picture suggests - the start and ending frames for this animation are not lining up well with each other, which is why the curves are red. Optimally, we can shorten the animation by moving the slider up to where the best quality match is on the curve, to reduce this problem:



What's great about this is if we look at the other curves, most of it matches well within the green even. All except for the horizontal (XZ) root transform position option, so we won't be baking that into pose for this animation. Additionally, as the length slider indicates, the animation is significantly shorter. Here is the result (just ignore the little loop hiccup, capturing perfect gifs is very "gifficult"):



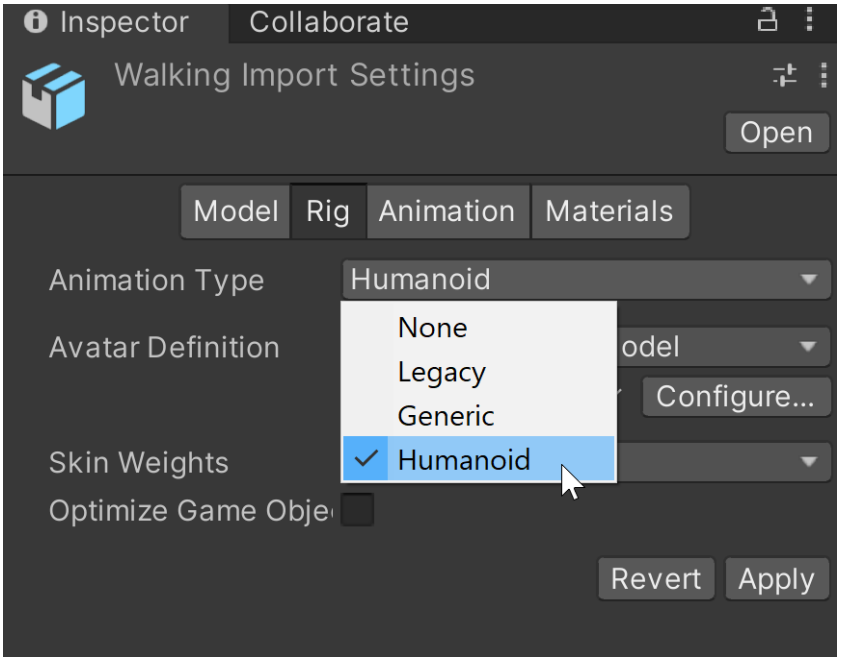
([PLAY GIF])

If you are in the camp that went with the 'duplicate animation from FBX' solution, you'll have to duplicate it again after making changes like this. Simply duplicate the animation from the FBX-file again, then replace the file we duplicated earlier with this new version. Rinse and repeat for all of the animation clips, until all of them look good and loop without imploding into humanoid spaghettis.

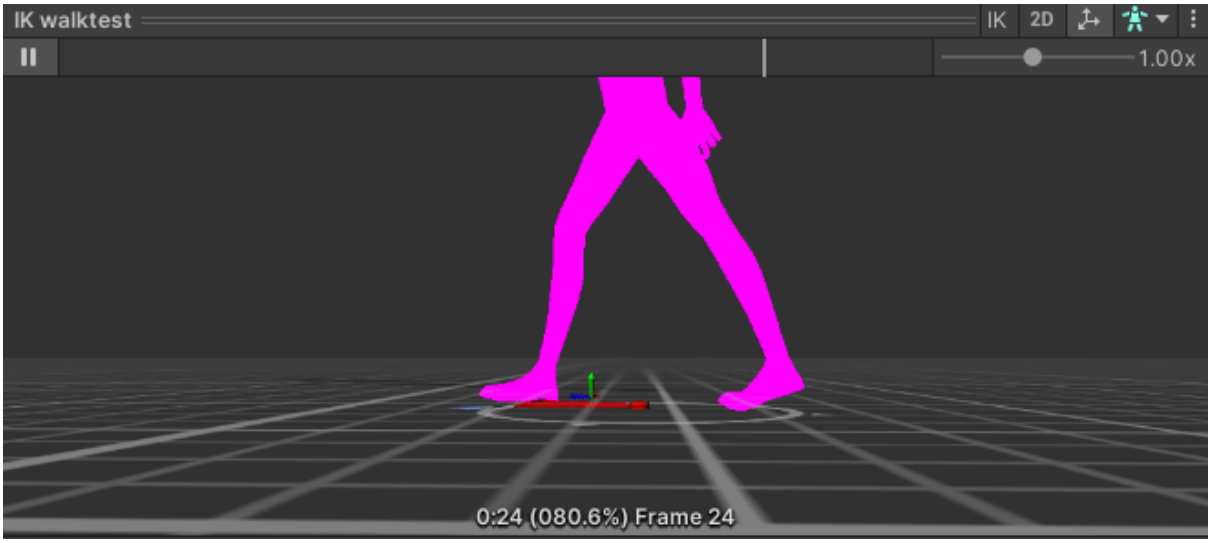
Protip: Try to keep looping animations such as walking and running, fairly short. This makes it *much* easier to implement logic on top of them later, such as dynamically correcting foot placement with Inverse Kinematics - which is something we'll be doing later as well.

4.1.3 Modifying animation rest poses:

Sometimes the animation clips themselves have alignment issues, even when we've gone through all the steps above. This doesn't happen with the samples we've been using so far, but just to illustrate the process in case it happens to you, I've grabbed a walking animation from Mixamo where this is a problem. As before, make sure your animation is retargeted to the Humanoid Avatar system for this:

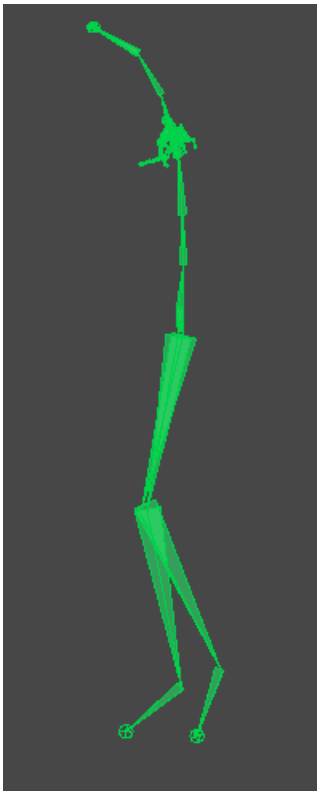


Next, go to the **Animation** tab and check the preview. If it looks off like in the gif below, it is due to the retargeting sometimes being a bit quirky, as the resulting rest pose can be something that isn't actually in a T-pose:

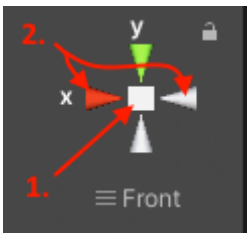


([PLAY GIF])
Got some swing in the step!

To fix this problem, we'll have to modify the rest pose of the avatar that was generated for this animation. Go back to the **Rig** tab and click the **Configure** button to open the avatar settings for this animation. Here is a side-view of the skeleton, which illustrates the problem:

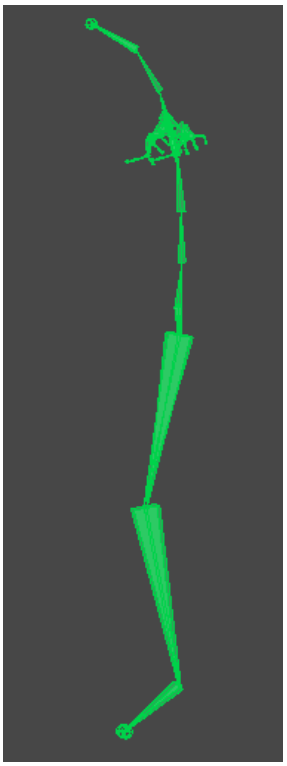

Whoopsies

To get this view, make sure you are looking along the X-axis in the isometric perspective:

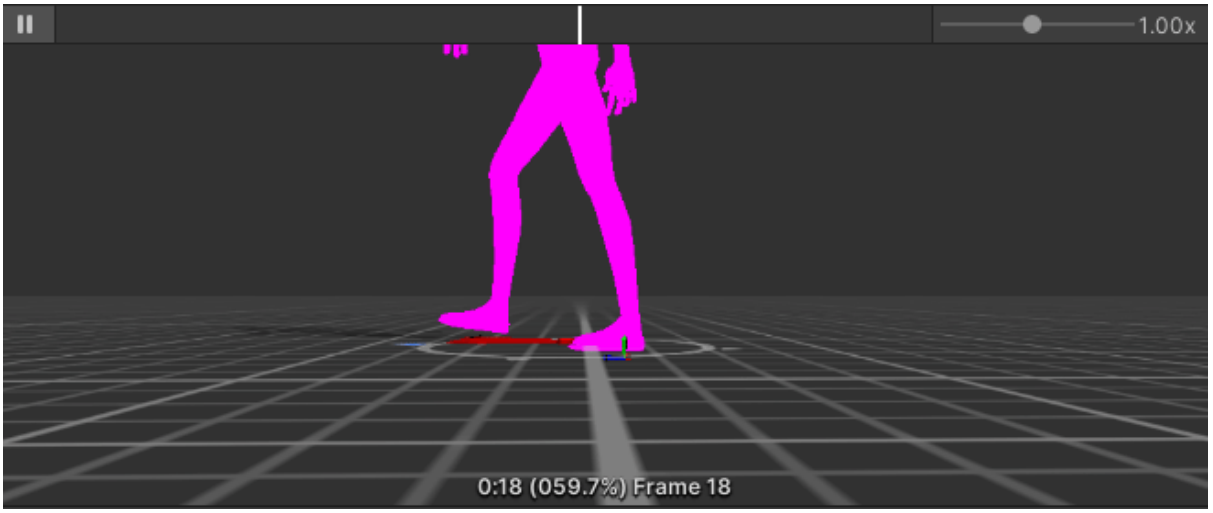


1. Click the white box in the middle to go to an isometric view.
2. Click on either of the X-axis arrows to the left or right on the box, to rotate to the left or right side-view.

Next you can - from the topmost bone of the leg and downwards - make rotation adjustments until both of the legs are in the same pose. You should of course rotate the leg that is causing the issues for this case and not the other way around. It doesn't have to be pixel perfect to look good, just get it to align up somewhat, like this:



Now if we look at the animation clip again, the foot looks much more normal:

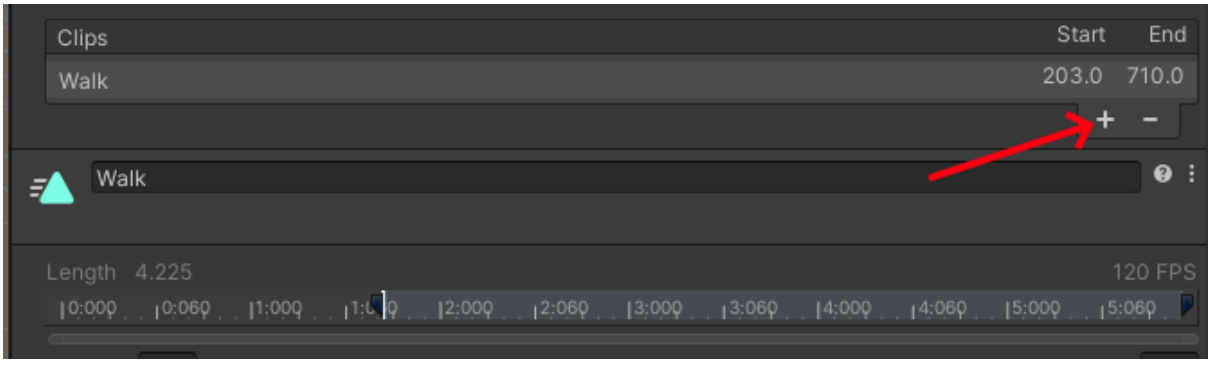


([PLAY GIF](#))

You can make further adjustments if needed, if you want things to line up differently (avoiding toes from clipping into the ground, for example), but this is good enough for most cases. This can be a slightly finicky process as it depends on the individual animation, but minor alignment issues are often fixable with inverse kinematics. I recommend that you pause the animation and look at it at a slower playback speed (or use the scrubber at the top to scroll through the frames), to get a better idea of how the misalignment issues are affecting the movement and what you might want to move to fix it. It could even be something like the legs being rotated differently when viewed from the front, for this particular case, so keep the different perspectives in mind when troubleshooting this process.

4.1.4 (BONUS) Making multiple clips from one FBX-file:

While we skimmed over the import settings in the Animation tab earlier, one of those settings might be pretty interesting if you have complicated and longer clips, but want to extract multiple, smaller animations from it. Instead of importing the same FBX multiple times, it is possible to just create multiple clips - even without duplicating the clips out of it. If we look right above the point where we changed the name of the clip earlier, we'll see this:



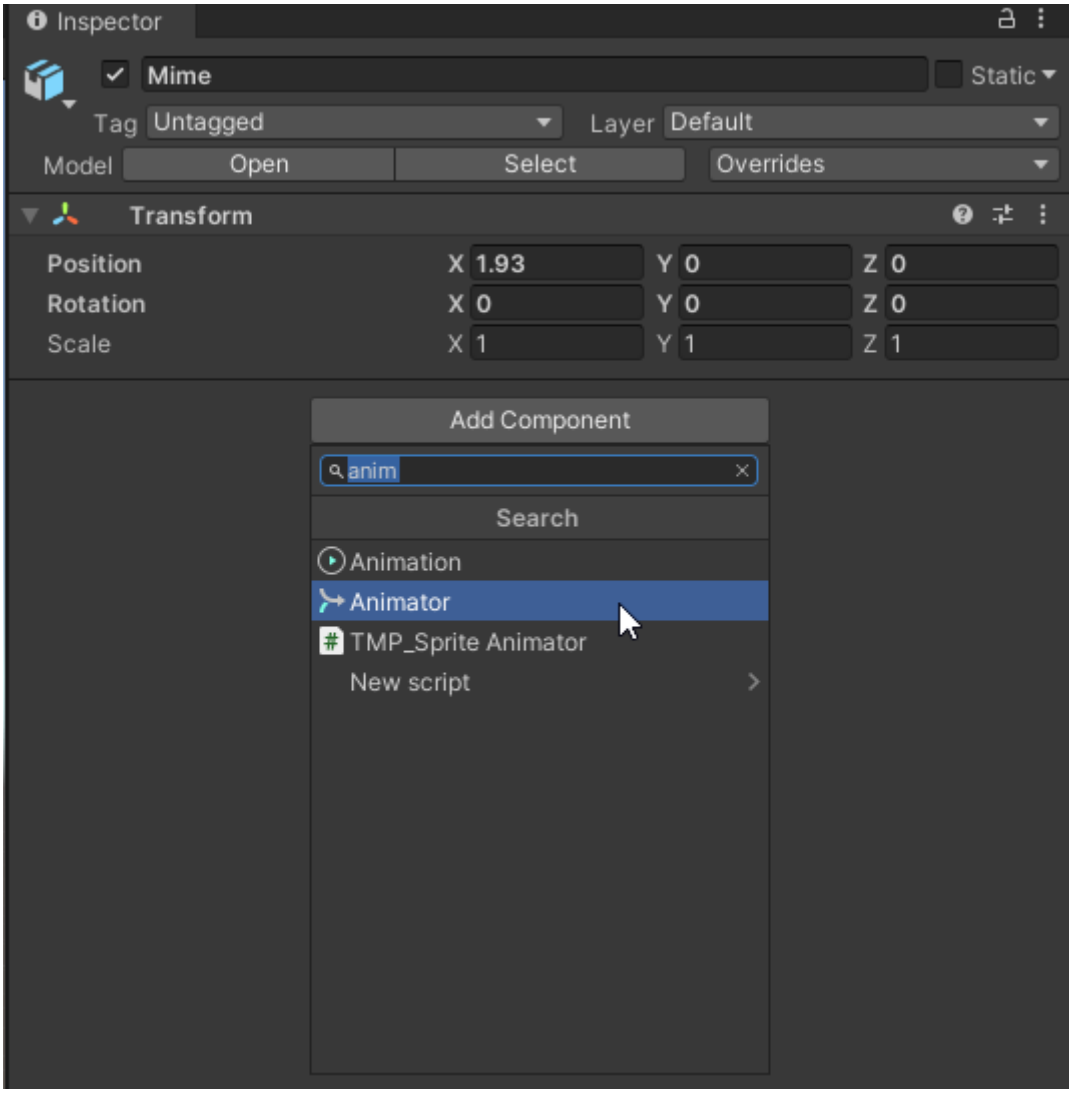
If you click the **+**-symbol, a new clip will appear in the list. If you click it, you can define its own properties (such as different start- and end-frames) and give it a different name. And that's all there is to it!

4.2 The Animator Component (and Animation Controller)

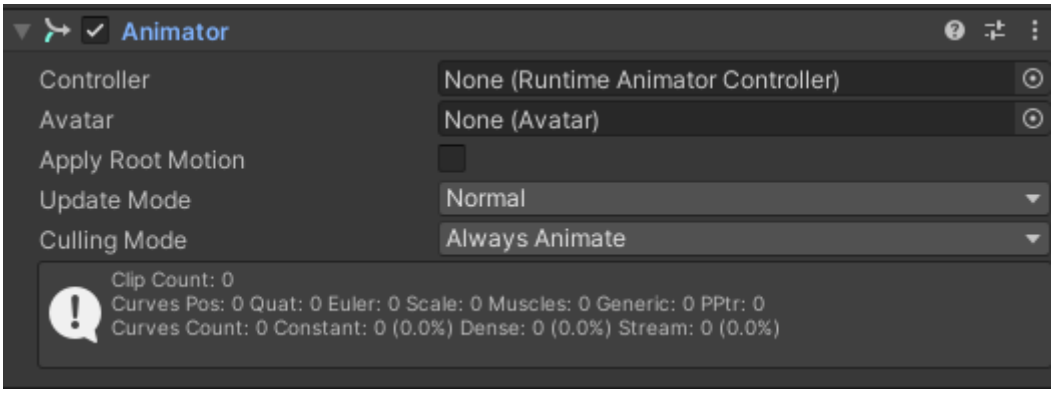
With the animations ready, we finally get to the implementation and logic that dictates when and how they are being played! This is done by adding an Animator component to the character, which in turn uses an Animation Controller, which is where we dictate what animations should play and set up transitions and logic to make sure they are contextually appropriate.

4.2.1 The Animator Component

Let's begin by adding the Animator component to the character we wish to animate. This is done by selecting the character - preferably the prefab version in the Project view (so that the changes apply to all instances of the character, not just the one in the scene). Please note that this is added to the topmost object of the hierarchy in the Mime object - the one that contains all of the other objects defining the Mime character:



The animator component has a few different properties to go through once added:



Going through the properties from the top, we've got:

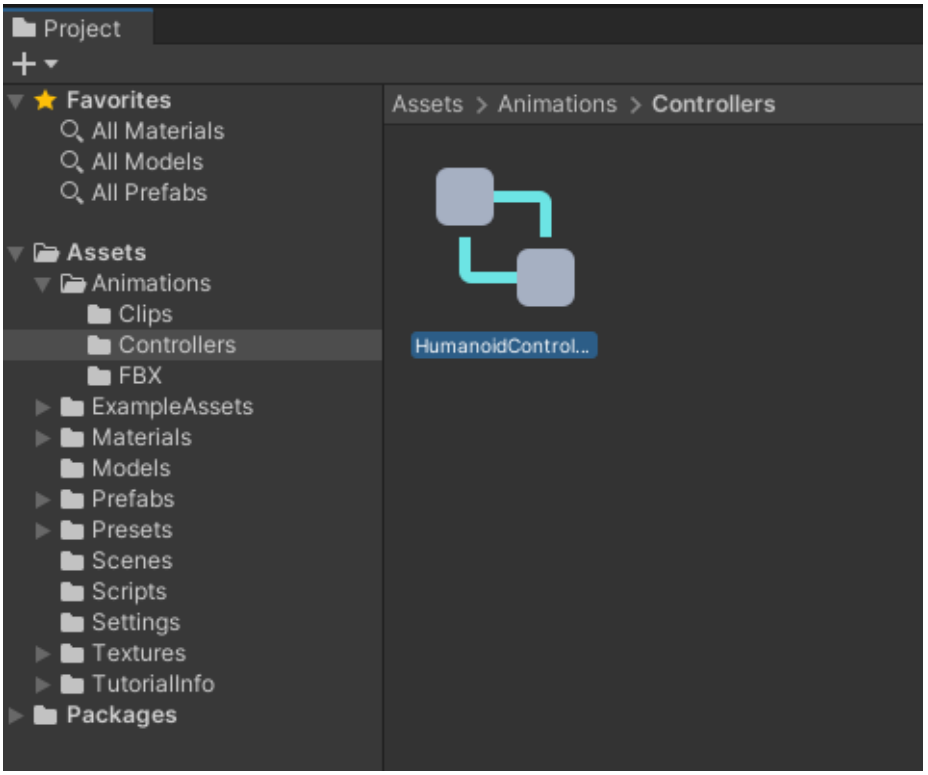
- **The Animator Controller** which I'll go into detail with in a moment
- **The Avatar** being used - which will show the avatars that we made previously when retargeting to Humanoid. Just pick the Mime one that you made (or whatever your character name is).

- **Apply Root Motion** dictates whether or not the animation is allowed to displace the character in general. Setting it to off can be great for providing movement control via a script instead, but setting it on allows the animations to drive the motion instead - in which case the player controls the character by controlling what animations are played, essentially. Choosing one thing over the other depends on how your different systems are supposed to work. For tight controls, favoring animation driven movement systems can be difficult to justify, unless the animations and the feedback from the controls are precise and responsive enough to make up for it.
- **Update Mode** is used to determine when the animation should update, as well as what timescale should be used. *Note that 'timescale' mentioned here means 1 = 100% speed, 0.5 = 50% speed, 0 = 0% speed.*
    - **Normal** runs in sync in the Update call each frame - the speed of the Animator matches that of the current value of timescale. *Use this if you don't need the character to interact with physics (eg. pushing boxes).*
    - **Animate Physics** runs in sync with the FixedUpdate, which is used for physics simulations, making sure that the physical interactions are taking the movement into consideration and vice versa when applicable. The speed of the animation still scales with timescale. *Use this whenever you want to interact with physics in your animation.*
    - **Unscaled time** does not care about timescale, it will always update at full speed. This is useful for stuff like UI elements, as you don't want your button animations to go in slow motion when the game does, for example.
- **Culling mode** is used to determine when the animation no longer plays; how it should stop and how it should resume.
    - **Always Animate** means the animation never stops, even when the character is not in camera. This provides good continuity but costs whatever processing power the animation clip requires to continue playing and updating the transforms of the character.
    - **Cull Update Transforms** stops any active retargeting, inverse kinematics and updating of transforms from happening, while the character is outside of the camera. The animation clip will continue 'rolling' in the background however, meaning once the character reenters view it will play as though it had kept playing.
    - **Cull Completely** simply means that everything stops while outside of view, meaning the animation will resume from the frame it had reached, when it reenters view. Eg. if a character jumps and you look away while it is in mid-air, it will remain in mid-air until it is being observed by the camera again.

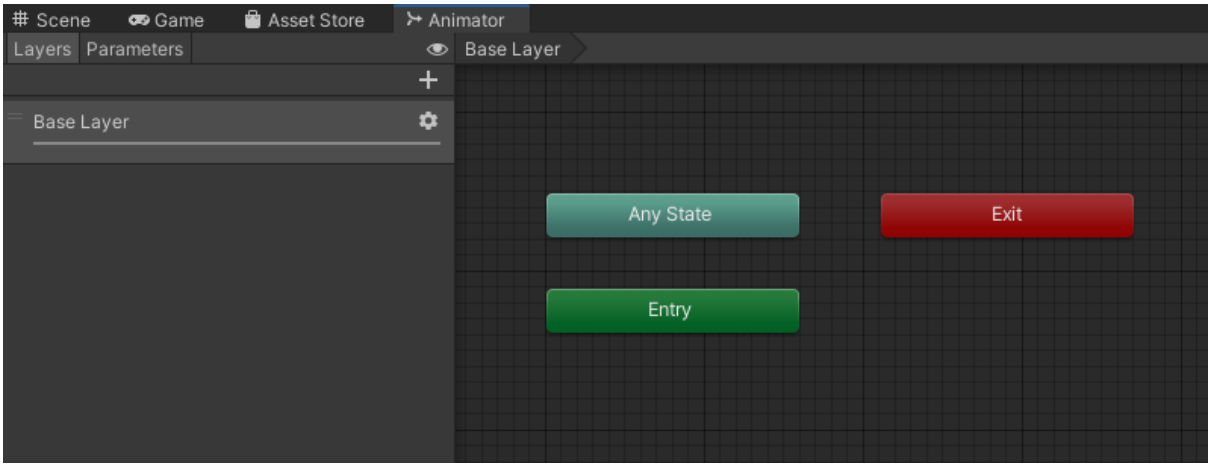
4.2.2 The Animation Controller

Before we can choose what Animation Controller we want to use in the Animator component, we will have to create one first. If the Animation Controller that you make is only meant to be used for one character, I prefer naming it something like '*<character name> Controller*', but otherwise I would be more generic with the naming - say if you want the same animations to apply to an assortment of characters, I'd rather write '*<character role> Controller*'. It should always be possible to deduce what the Animation Controller is being used for in your systems, by just looking at the name. So for this case I'd either make a

generic '*HumanoidController*' for animating both characters, or a specific '*MimeController*' and '*VikingController*' for implementing different logic in each of them. Either solution works for this tutorial as we will only be setting up one controller anyway, but it is important to be aware of when these distinctions should be made in your projects. I just went with the Humanoid solution this time:



If we double-click on this Animation Controller, we'll be greeted with the Animator window, which should look something like this (though probably less compressed, I've zoomed in to make it readable):



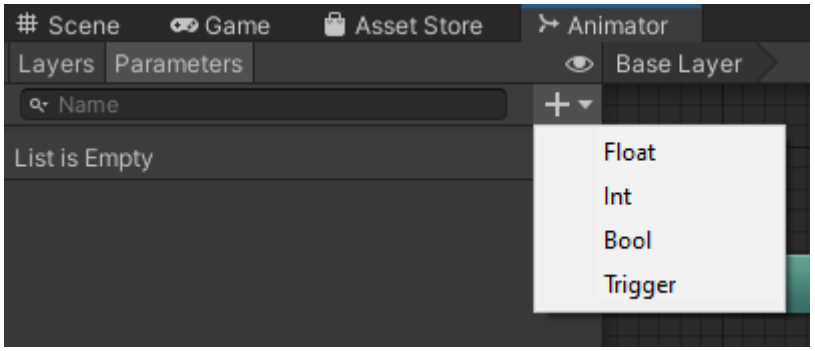
For now, let's just acknowledge that there are two tabs in the upper left corner, one for **Layers** and one for **Parameters**, as well as a view on the right side showing the currently selected layer. The available layers can be seen on the left side, provided that the Layers tab has been selected.

4.2.2.1 Layers (basics):

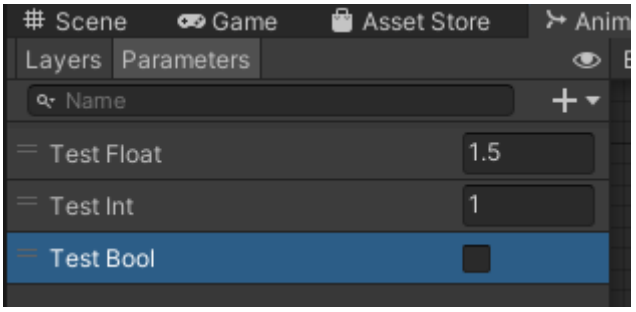
For now, let's just acknowledge that layers are a way for us to have several animations running simultaneously, with some logic to which one takes precedence over the other - great for characters having to do multiple animations at once (like running and shooting at the same time). But we'll start with just one layer, then get into more complicated interactions later.

4.2.2.2 Parameters:

Parameters are variables that we can easily modify from elsewhere. If we click on the Parameters tab, we'll be greeted with an empty list. Click on the little **+**-icon with the arrow next to it, to see what options are available for variable types to be made:



If you have a bit of familiarity with programming, you'll know what the first three options are (if not, I recommend you go check out the crash course I linked at the beginning of the article). Selecting either of these will create them in the list, allowing us to name them, like this:

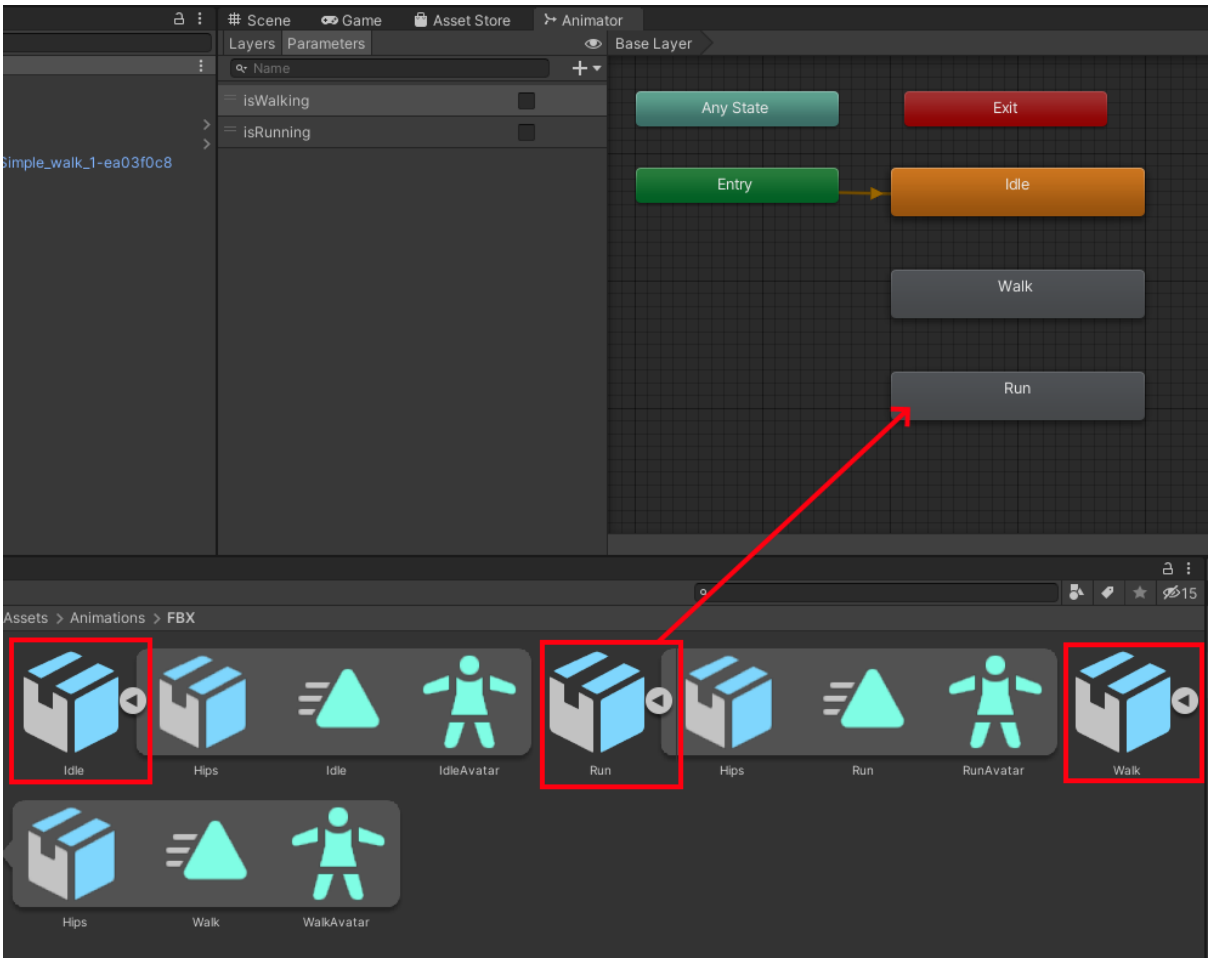


The last type called **Trigger** is a lot like a bool in that it can either be true or false. However, it automatically reverts back to false after reading true once, which makes it useful for creating transitions between your animations, without having to think about resetting the logic that you used to trigger the transition.

4.2.2.3 Layer Editor:

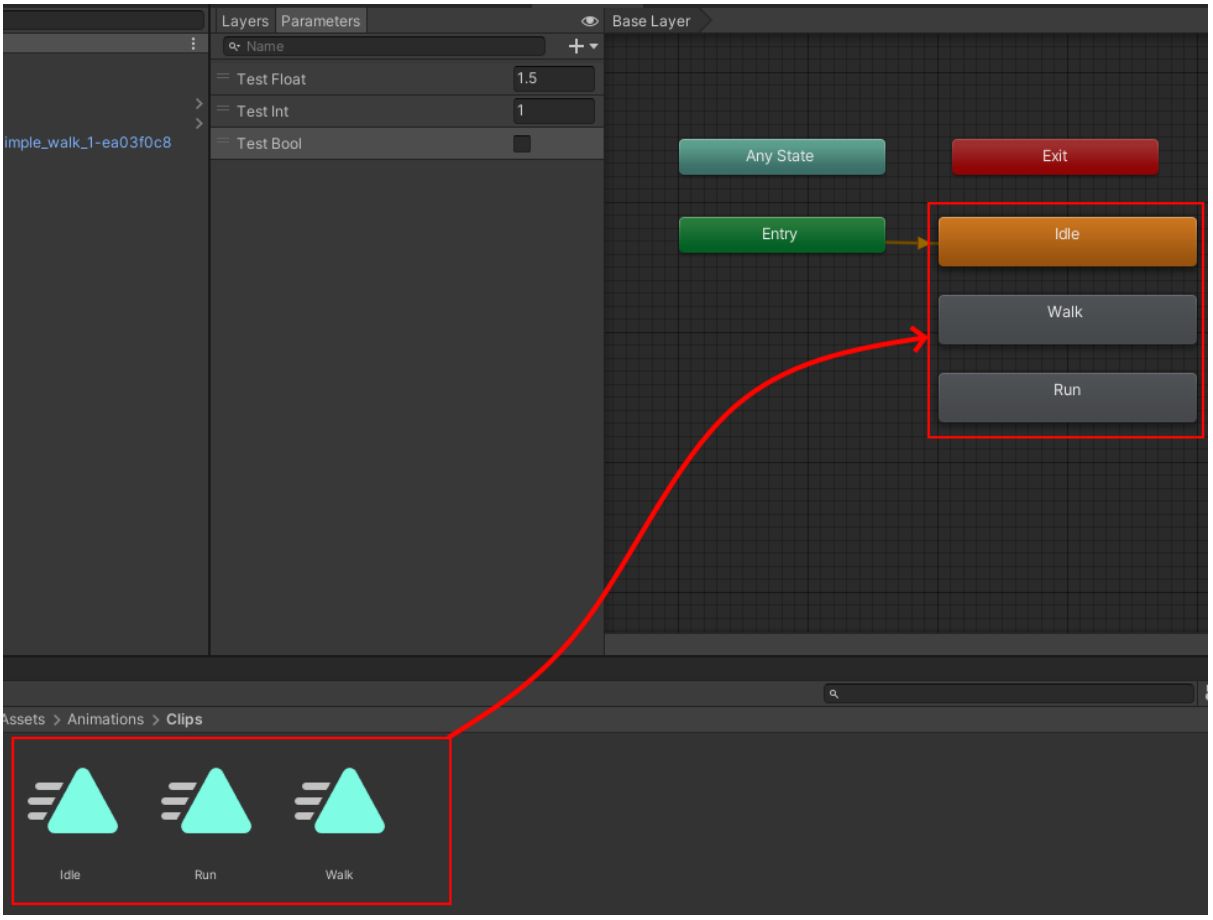
The Layer editor is the view on the right side of the Animator window, where the three coloured blocks saying '*Any State*', '*Entry*' and '*Exit*' can be found. These blocks represent states that the controller (and by extension, the character it controls) can be in. By creating new states, each of which dictates what animations are played when those states are active, we build the logic for our animation system.

Creating a new state is easily done by either right-clicking in the layer editor and selecting **Create State -> Empty** or as I recommend that you do for this tutorial, simply drag your FBX file into the layer editor like this:



Note that if you have multiple animation clips in your FBX files, this will result in an individual state being created for each clip.

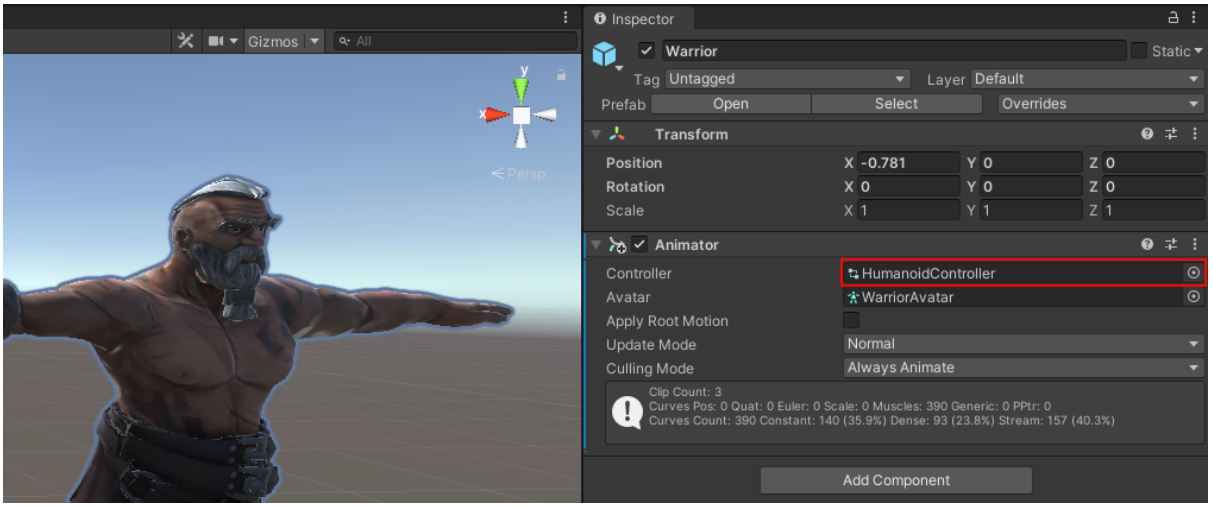
Or alternatively if you duplicated the animation clips, you can just drag those in directly as well:



Notice how there's an **orange arrow** from the **Entry** state to the **Idle** state? This is due to the Idle clip being the first one that I dropped in there, but what this shows is that *the Animation Controller always starts at the **Entry** state,* then moves on from there. That means when this animation controller is activated (eg. loaded into a scene), the Idle animation will be the one that plays.

If the Idle state is not the one that your Entry state is pointing to, just right-click the Idle state block and select **Set As Layer Default State**.

Before we move on to how we can construct and play around with these logic systems, go back into the scene, select your character and add the newly created Animation Controller to its Animator component like this:



With this set up, pressing the play button will show the idle animation playing, looking something like this:

(PLAY GIF)

With the basics out of the way, let's move on to constructing the logic for animation transitions and the different ways we can do this.

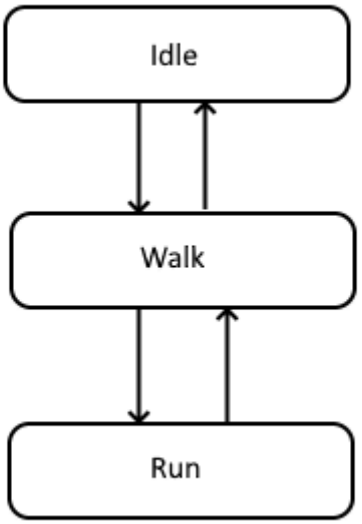
4.3 Building the state machine:

While this chapter continues to explain how the Layer Editor in the Animation Controller works, setting up the logic that dictates how the character should be animated, branches out into a few other systems (most notably the one we'll write ourselves in code).

But before we do any of that, I recommend that you familiarize yourself a bit with the way these logic systems are designed at the conceptual level, as shown here in Unity's documentation: https://docs.unity3d.com/Manual/StateMachineBasics.html

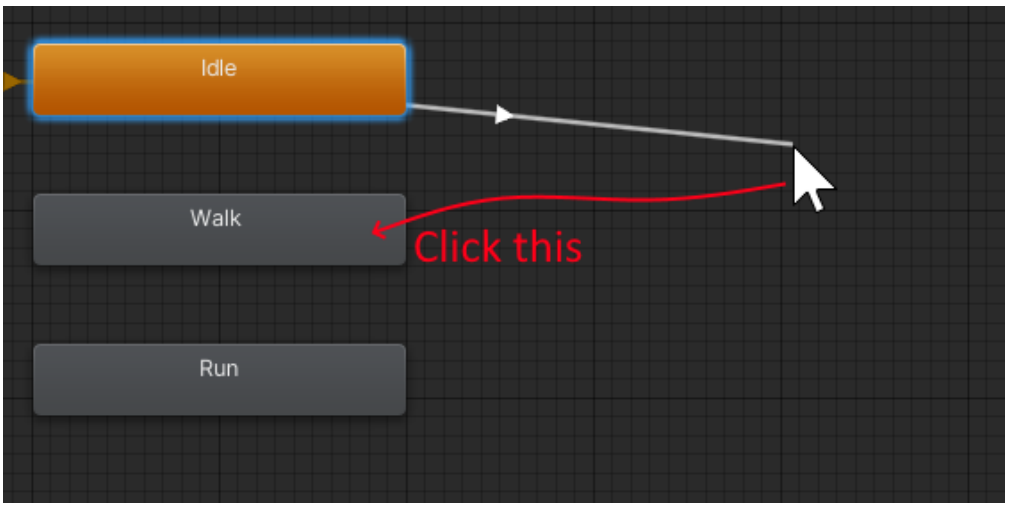
4.3.1 State transition setup:

The idea is to start simple, on paper even, trying to outline what your character is supposed to be doing and how they might logically go from one state to another. As illustrated in the article I've linked, it boils down to preventing actions from occurring, in contexts where we do not expect them to. To illustrate the concept, here's a simple overview of what I will be building for this tutorial:

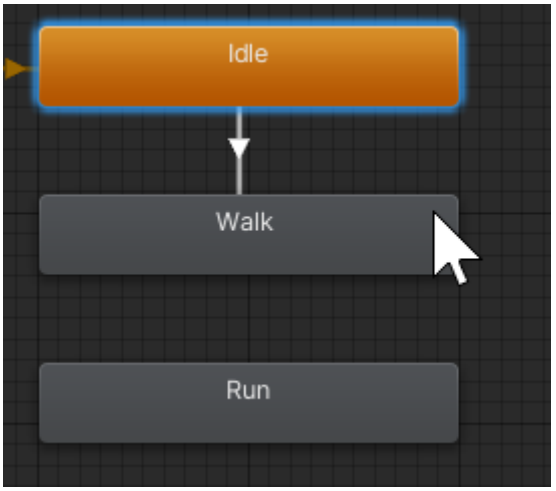


The idea is that the character will never be able to transition directly from Idle to Run or vice versa, without blending through the Walk animation first. This gives a more natural transition between the states. We can still speed up the transitions to make it almost instantaneous, which is necessary for very responsive controls, but figuring out what makes the most sense for your characters and animations is an iterative process: It all boils down to testing and making modifications until it looks and feels right.

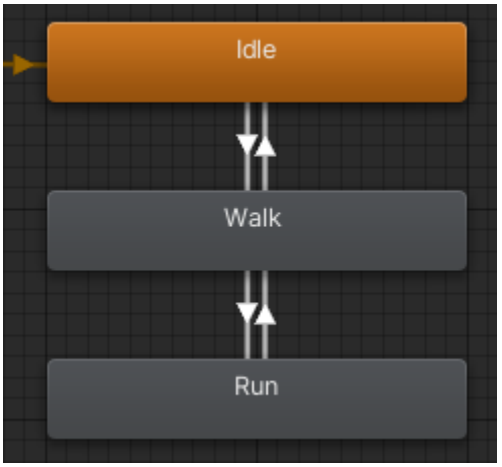
To add new transitions (arrows) to your animations, right-click the animation state and select **Make Transition**. This will show an arrow from the animation state to your mouse, which you can snap to the state you want to make the transition to, like this:



The arrow will snap to the other animation state like this:

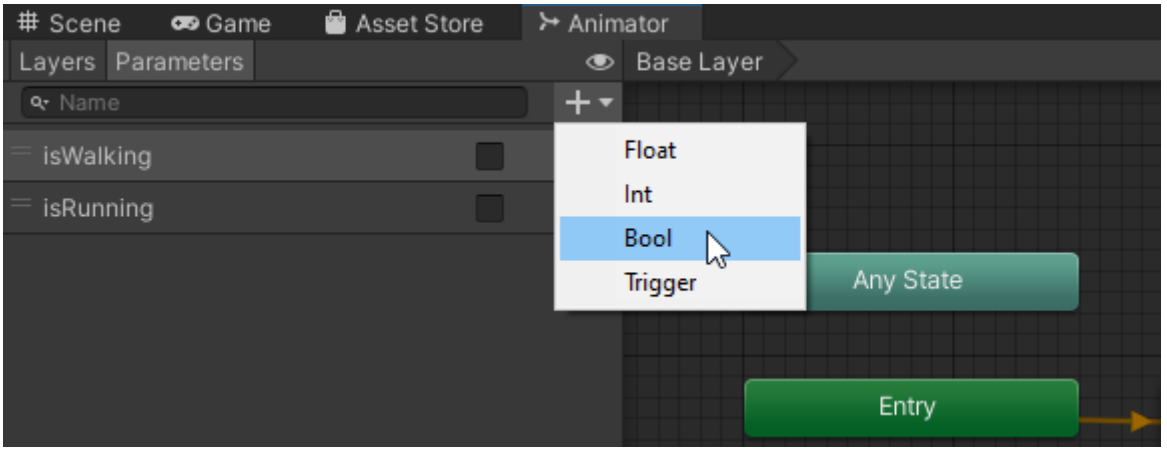


Keep repeating this process of adding transitions until it matches the initial drawing:

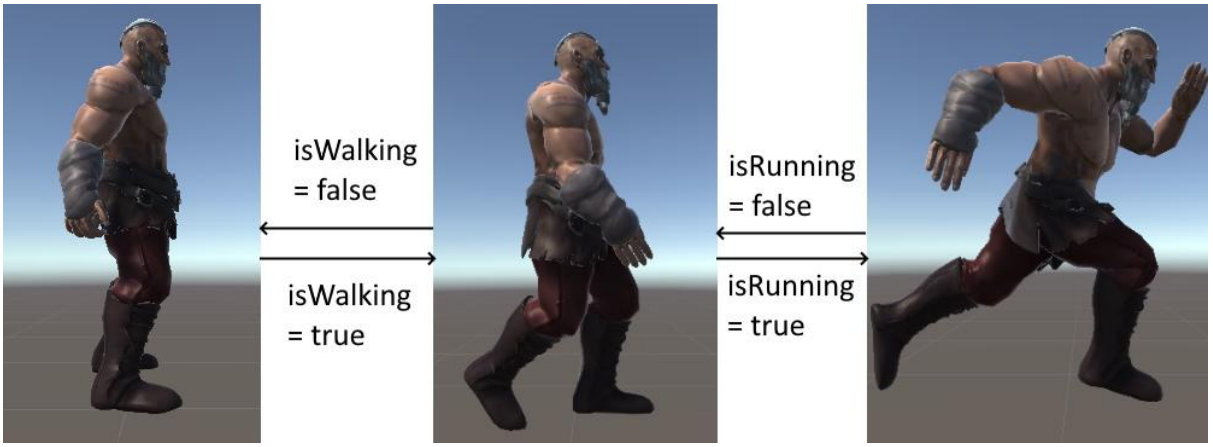


With the transitions set up, we need to think about what we might use to indicate when the transitions occur. Thankfully, for this setup it is quite easy - just make two Bools to indicate if the character is walking and if the character is running. We'll make one called **isWalking** and

one called **isRunning**. Go back to the Animation Controller in the Parameters tab and add them:

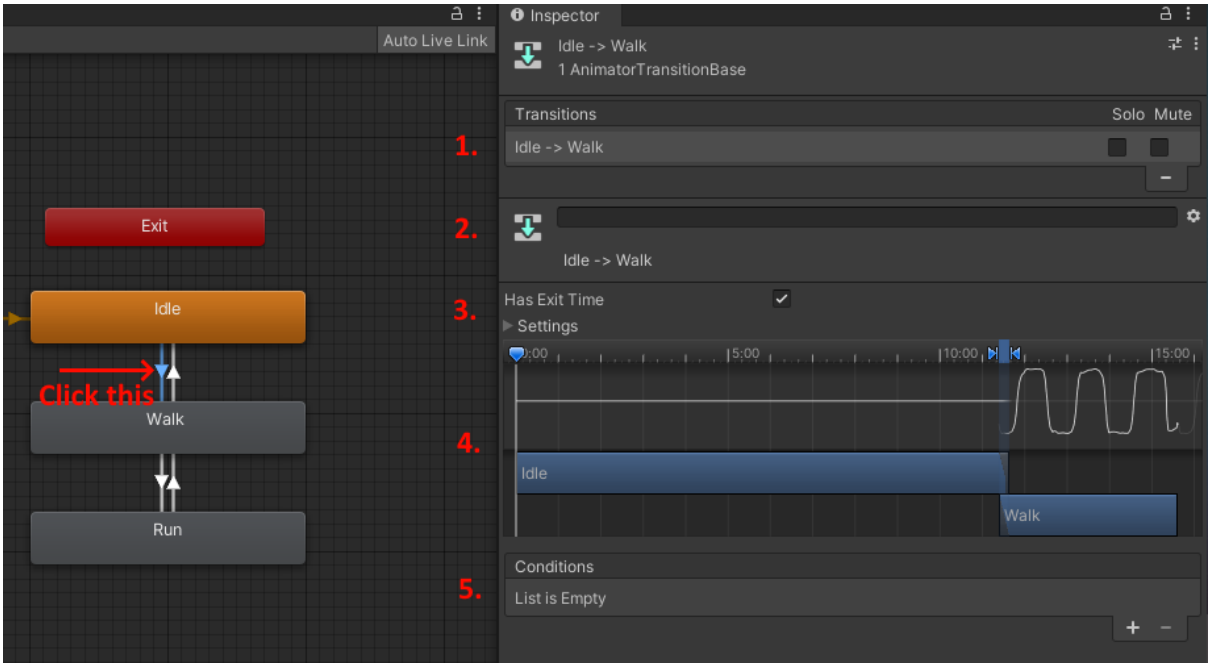


Here is an illustration to showcase why these two checks make sense:



Remember that a bool is either true or false - either the character is walking or not - either the character is running or not. That means we don't need a third bool to indicate if the character is standing still, because we can assume that if they are neither walking or running - (both of the bools are false) then they must be standing still.
We'll be exploring this logic a bit more when we write some simple code later (for example - if the run button is held but no direction is given, then the character should still be standing still). For now, it is enough to determine what indicators are required to move the animation state around, then the time they are triggered from the player input, is determined by the programmer (or you, doing the programming, which also makes you a programmer!).

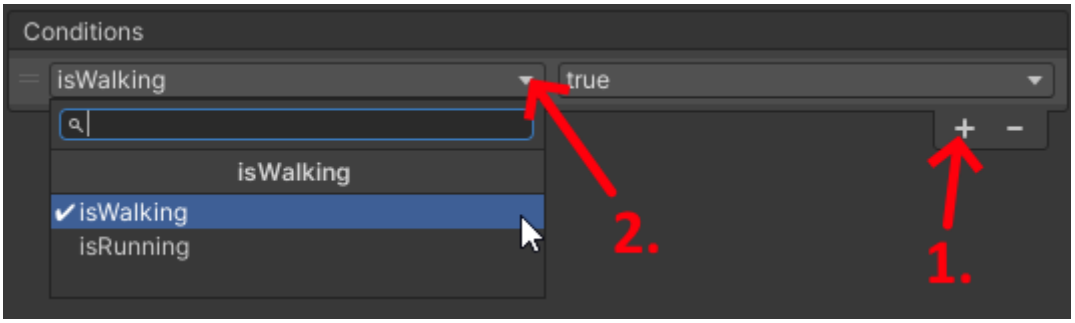
The next step is making use of these bools in the transitions between animation clips. The transitions can be accessed by clicking on them (the white arrows between the states) directly. The view for a transition with default settings will look something like this (minus the red text):



I've added numbers to explain each part that you are seeing here. We won't be using all of it, but you should have an overview of the tools available:

1. **Transitions** indicate if it should be possible to transfer from one state to multiple other states. We won't be using this for now, but if set to **solo**, only that state should be an option, or if **mute** this state is not an option and the transition to it won't happen. This is useful if some states are only available/unavailable during specific points in the character logic (so your character doesn't play dead by accident or randomly climbs an invisible tree where there is none). Here is the official documentation for it.
2. This field lets us name the transition, making it easier to identify in other contexts. I recommend calling it something like 'Idle to Walk.'
3. **Has Exit Time** is a setting used for timed transitions. You should toggle this off for now, as it is a timer for animations which transition to the next state after the timer ends. There is an expandable assortment of settings that you can view here, related to the exit time functionality, but I'll refer to the official documentation for a deeper explanation of these.
4. This is the **timeline** that shows how the animations are blended together. The blue *Idle* and *Walk* bars indicate the animations and their lengths. The two blue arrows pointing at each other indicate the beginning and end of the transition - if these are close then the transition is short and vice versa if the distance is long.
5. **Conditions** are where we can use our bools to trigger transitions, instead of relying on time.

For this setup, we will mostly be using the **timeline** and **conditions** sections. Let's start by setting up the conditions. Clicking on the transition from the *Idle* to the *Walk* state, then clicking on the little '+' icon in that section, we'll add a condition to the transition.



Once we've set up all four transitions as illustrated with the pictures of the Viking character above, we need to move on to our first basic code, to control what animations should be played.

4.3.2 Writing the basic logic:

As a heads up, this chapter is written with the assumption that you have a bit of familiarity with Unity and C# already. I will only be explaining the parts which relate to the animation controller, so I recommend checking out the introductory Unity video at the start of this document, if you are completely new to either Unity or C#. You can also use it as-is without knowing exactly what the code does, but it's nice to grasp the basics of it.

I am going to start by just pasting the functional code and then go through it:

```
using UnityEngine;

public class AnimationStateController : MonoBehaviour
{
    Animator animator;
    int isWalkingHash, isRunningHash;

    // Start is called before the first frame update
    void Start()
    {
        animator = GetComponent<Animator>();

        // StringToHash used for performance reasons
        isWalkingHash = Animator.StringToHash("isWalking");
        isRunningHash = Animator.StringToHash("isRunning");

    }
```

```
    // Update is called once per frame
    void Update()
    {
        bool isRunning = animator.GetBool(isRunningHash);
        bool isWalking = animator.GetBool(isWalkingHash);
        bool forwardPressed = Input.GetKey("w");
        bool runPressed = Input.GetKey("left shift");

        // From standing still to walking
        if(!isWalking && forwardPressed)
        {
            animator.SetBool("isWalking", true);
        }

        // From walking to standing still
        if(isWalking && !forwardPressed)
        {
            animator.SetBool("isWalking", false);
        }

        // From walking to running
        if(!isRunning && (forwardPressed && runPressed))
        {
            animator.SetBool("isRunning", true);
        }

        // Stop running if forward or run are no longer pressed
        if(isRunning && (!forwardPressed || !runPressed))
        {
            animator.SetBool("isRunning", false);
        }
    }
}
```

Declarations:

I've declared an **Animator** which is the script component that sits on the character object. Through this, we can access the booleans that we made in the Animation Controller, as the Animator is using this as a parameter.

I've also declared two **Int**s, which we'll just use to store some data. I'll explain these in the Start function where we use them.

Start function:

I begin by fetching a reference to the Animator component so that we can use it.

Next, I initialise the two weird integers that we saw above. This means turning the names into numbers, which can be used to identify the booleans that we made, without having to enter the name as a string. Why? It just runs faster, which is great because we'll be using them up constantly.

Update function:

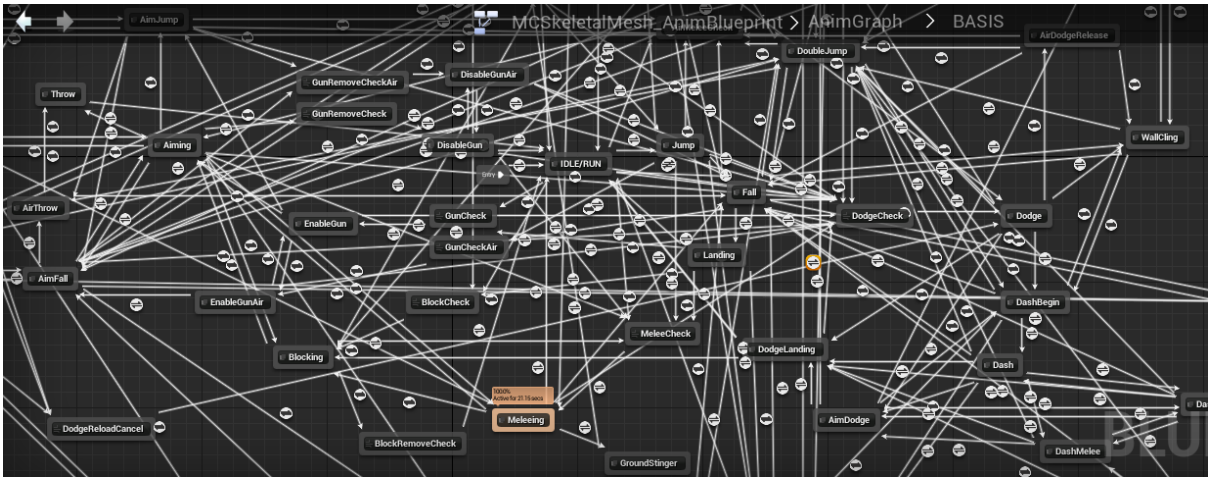
Here we simply get the status of the booleans, as well as check if the user is pressing the 'W' key or 'Left Shift' which are commonly used in games for walking forward and running. The rest is just a few if-statements to determine the results:

- If we aren't walking and W is pressed, **isWalking** becomes true.
- If we are walking and W is let go, **isWalking** becomes false.
- If we aren't running, but W and Left Shift are held down, **isRunning** becomes true.
- If we are running and either W or Left Shift are let go, **isRunning** becomes false.

The idea is simply that we can't run if we aren't already walking, so if we stop pressing forward while holding the run button down, the character should still stop running. You should of course write your own logic states for whatever makes sense for your game, this is simply to illustrate how you can access and change the values of your Animation Controller variables, from a script.

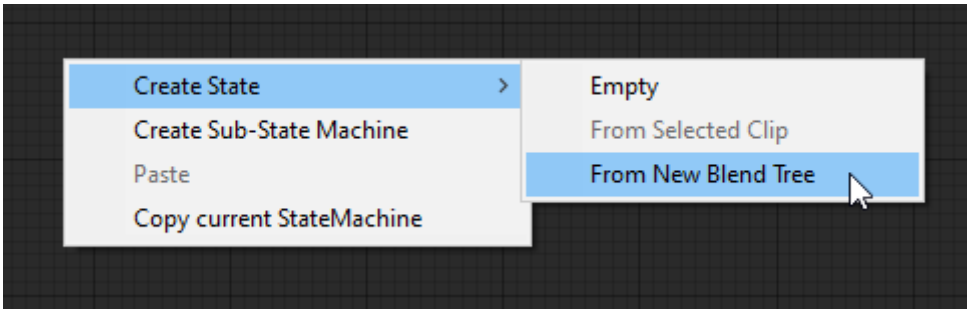
4.4 Blend trees:

With all of these different states and variables, it is easy to lose track of where we are in the state machine and where we can go. For simple characters and interactions, the above method is quick and easy to read, but for more complex systems - such as characters that are able to move in eight directions at different speeds, you can imagine how the state machines quickly turn into something you normally would be served at an Italian restaurant with meatballs and tomato sauce:

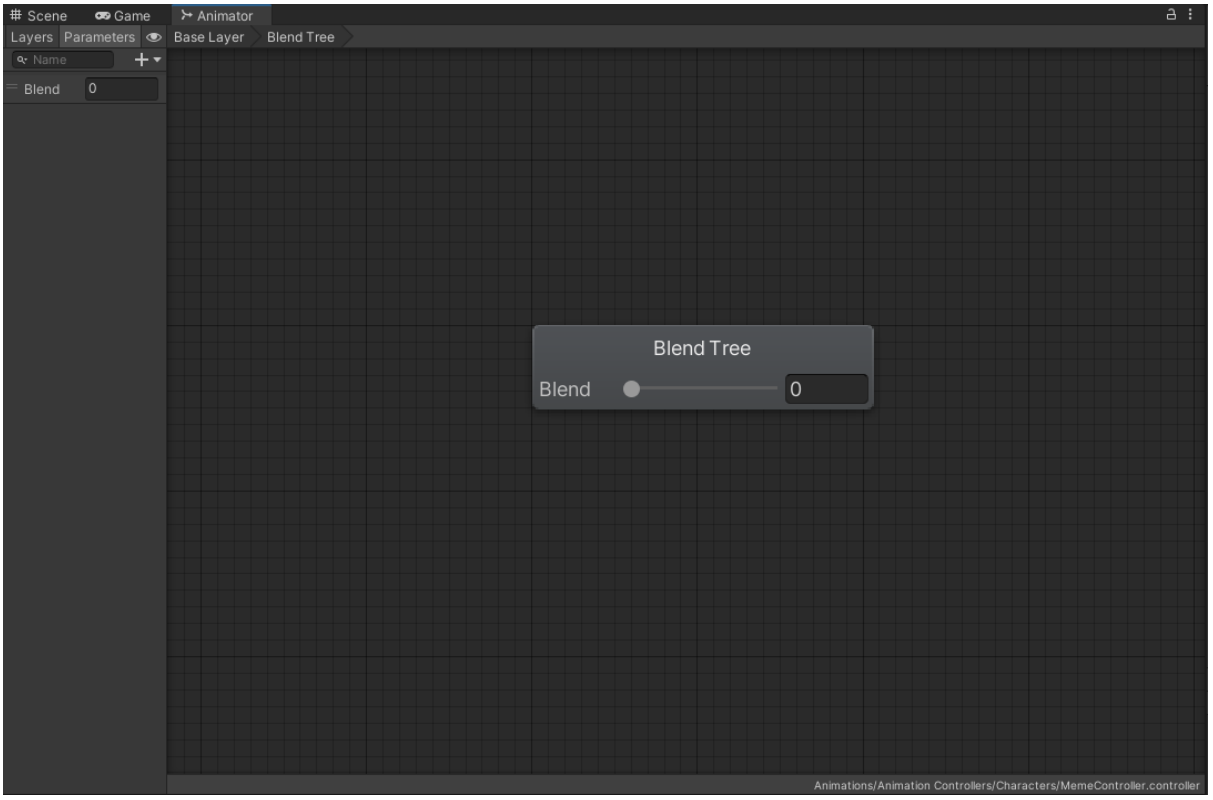


Picture from Unreal Engine when searching for "State machine spaghetti"

Thankfully, there are more than one way to perform these transitions - most notably the **Blend Tree** system (which you can read the official documentation on here, if curious).
To begin working with Blend Trees, we have to create it directly in our Animator Controller window, by right-clicking in an empty space and selecting **Create State -> From New Blend Tree:**

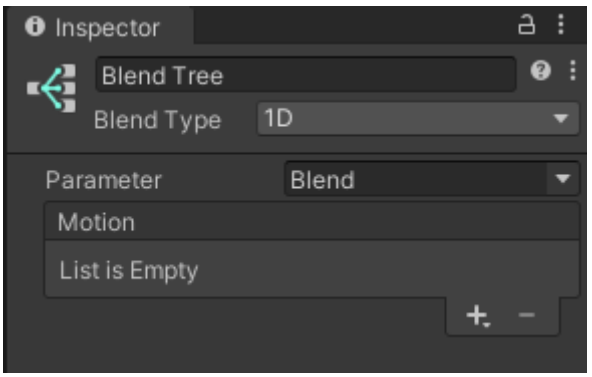


When the new State has been created, we can access the Blend Tree by double-clicking on the State itself, which brings us to a look like this:

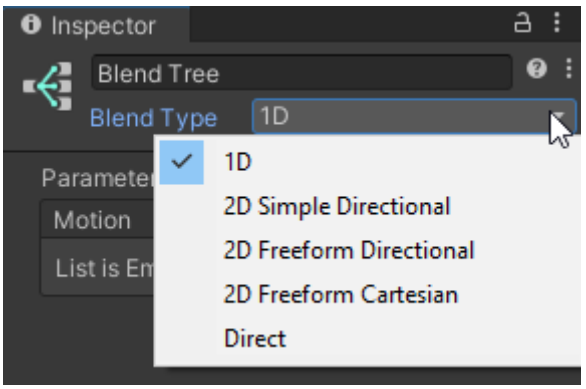


Notice in the upper right corner where the tab for this window is, that it now says Base Layer > Blend Tree. This shows us where we currently are within the hierarchy, as well as allows us to navigate back up through the previous layers, just as we would do with folders.

If we click the Blend Tree in the middle, we'll be able to view its properties in the Inspector window, looking like this by default (plus the animation preview that I have cropped out, as we already know this from earlier):

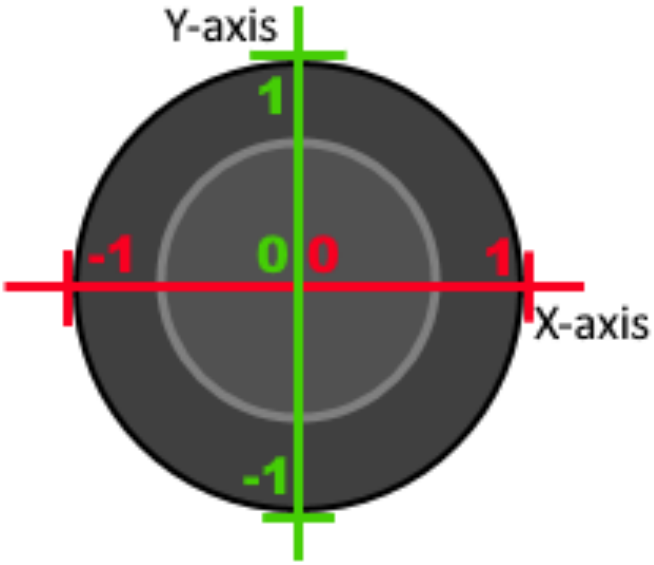


The only things that we should consider ourselves with right now, is the **Blend Type** and the **Motion** list. To elaborate on what the blend type is first, let's take a look at the options that are available to it:



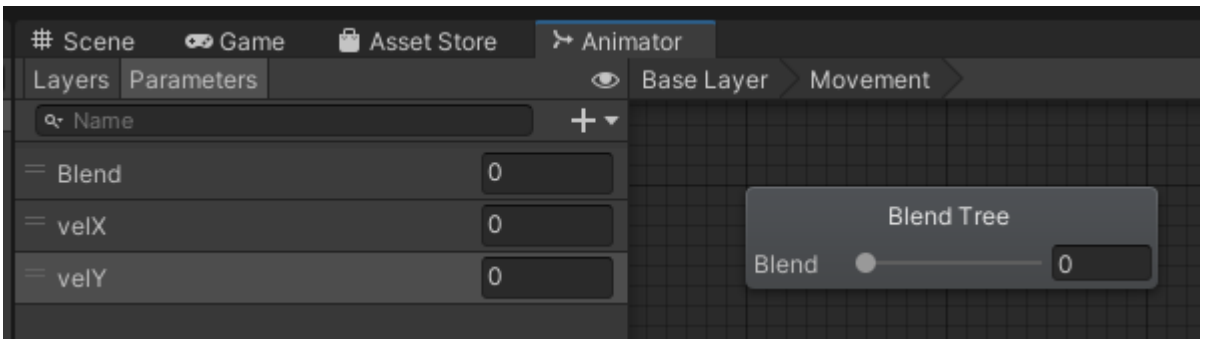
- **1D:** Meaning 1-directional, is just using a single value to blend between animations. For example: This is useful an object that animates differently at different speeds, but with no consideration for direction, only how fast it is going.
- **2D (all options):** There are multiple options in 2D, but I'll summarize them for movement on a plane - forwards/backwards/left/right, represented by two axes (usually X and Z, or X and Y, depending on the direction of the plane). If you want explanations for all of these, I recommend checking out the official documentation, but we will be using **2D Freeform Directional** for this, as it works well with both joystick and keyboard input. You may want any of the others if your inputs need to be interpreted in ways that work well with those options.
- **Direct:** As the name suggests, this allows for direct control over the blending values, which you can access and modify manually. Unity suggests using this for things like blendshapes (distorting meshes, most commonly faces but really any part of a mesh) and random idle animation blending.

As mentioned, start by selecting **2D Freeform Directional**, then add two float variables to the animation controller, which will represent the movement of the joystick on the X- and Y-axis, as shown here:

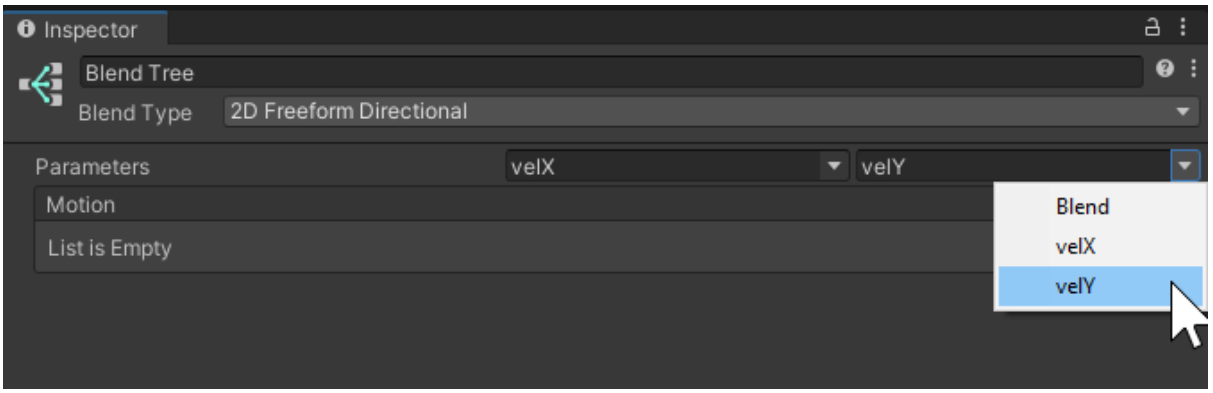


I'll explain more in detail when we make use of it - for now just acknowledge that the Y-axis is forwards/backwards and the X-axis is left/right.

I'll just name the two float parameters **velX** and **velY** to represent the velocity of the character in either direction:

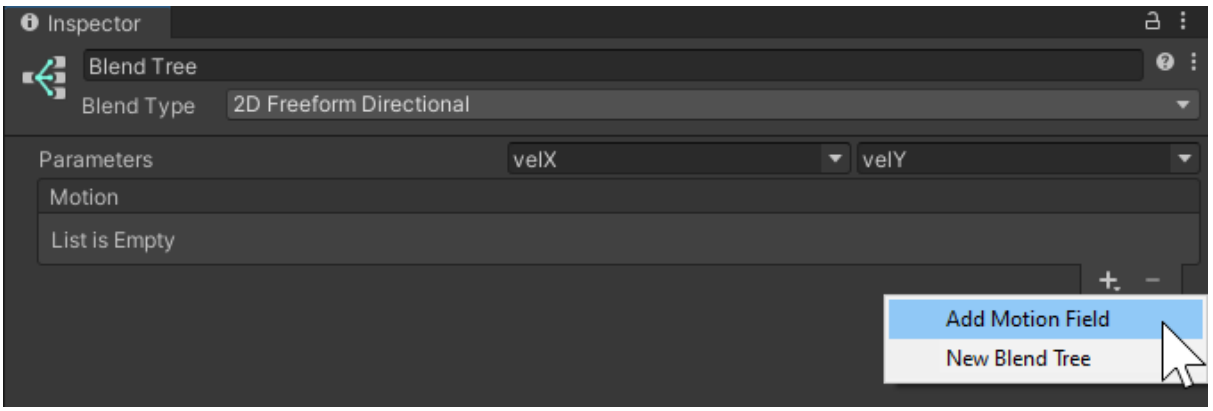


Ignore the 'Blend' variable for now. This is just a default float that Unity generates, we will not be using it, but we can't delete it until we've made sure that nothing in the Blend Tree is using it. To to this, click on the Blend Tree node to view it in the inspector, then change the parameters to use velX and velY instead, like this:

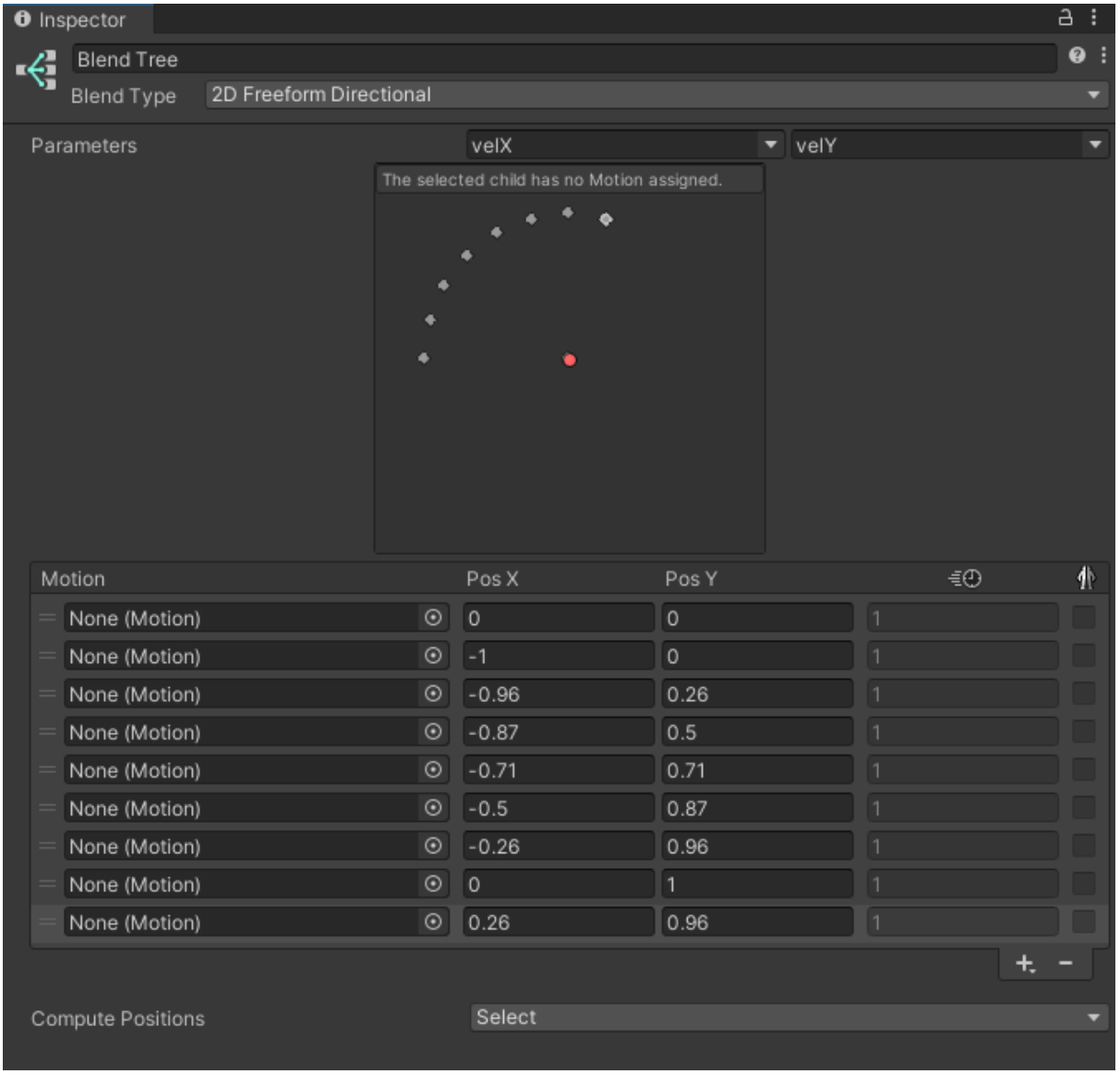


Now we can delete the Blend variable by right clicking on it and selecting Delete, as we no longer use it.

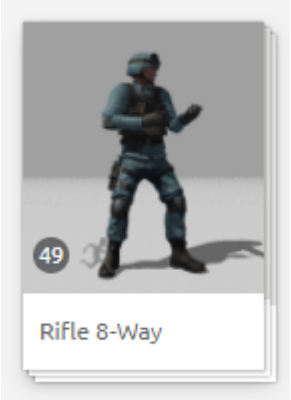
Next we'll want to populate our Motion list with different types of animations. It is common for characters to be able to run in eight different directions with a setup like this, so keep clicking the little +-icon and select "Add motion field" until you have nine motion fields (the ninth one is for the character standing still):



After you've added at least two, there's a new part of the UI that shows up, but keep adding new ones until you end up with a screen that looks like this:

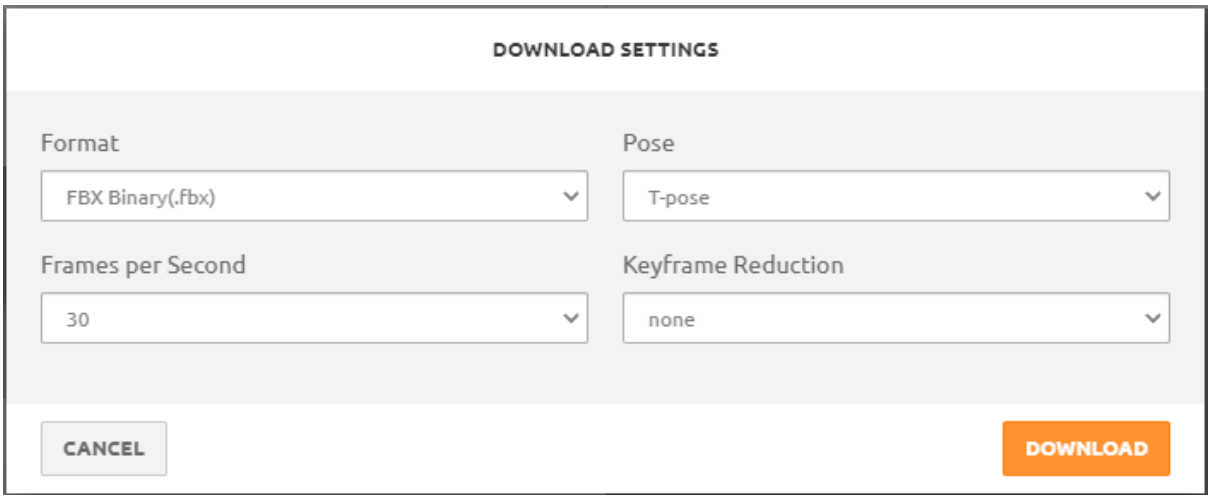


Before we get into what's going on here, we're going to need animation clips for all eight directions. The easiest place to find this for learning purposes, is to go to Mixamo and download this bundle of animations:



Just search for '8-way'

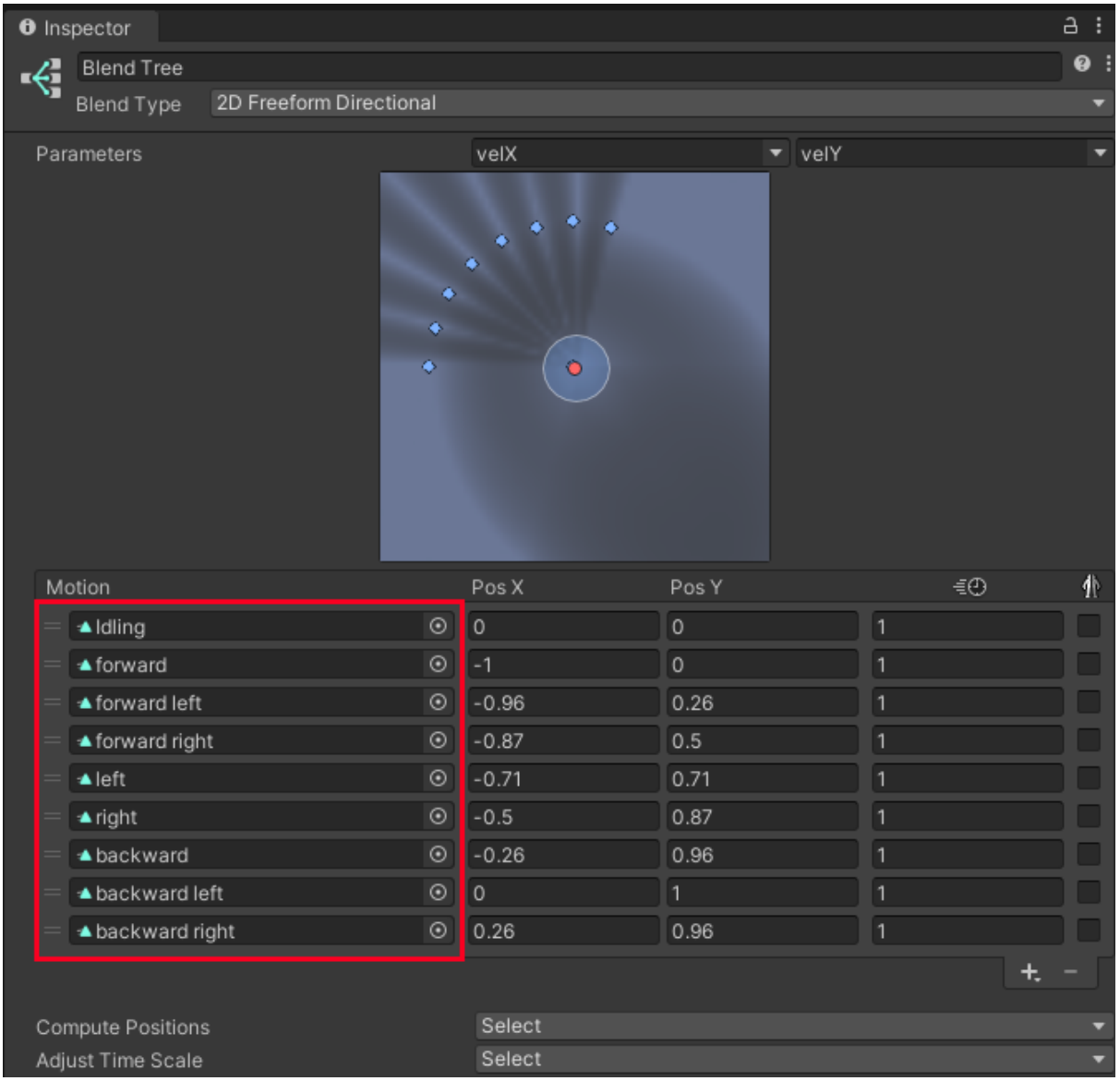
With these settings (the default settings):



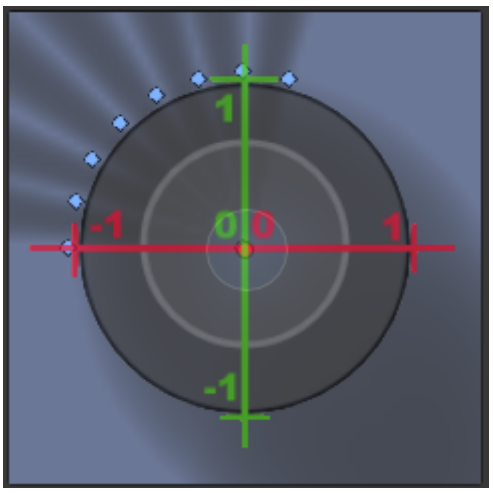
The animations I recommend using here are (you can use either the walk or run variants):

1. Walk right
2. Walk left
3. Walk forward
4. Walk forward right
5. Walk forward left
6. Walk backward
7. Walk backward right
8. Walk backward left

You can use any kind of idle motion, but the easy solution is to just grab the one called “idle” from the same bundle as well. Import each of these into the solution and make sure that you have retargeted them properly and that they loop. Here’s how I’ve set up the clips:

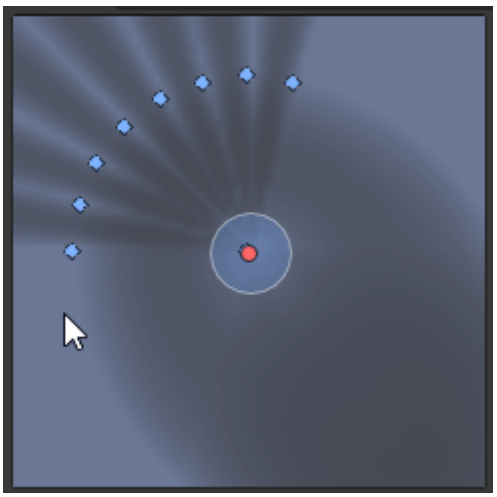


They don't have to be in this exact order, but that is just how I've set them up for now. To elaborate on what's happening in that box above the list of clips though, let's bring back the illustration of the thumbstick from before as an overlay, to understand what's going on:



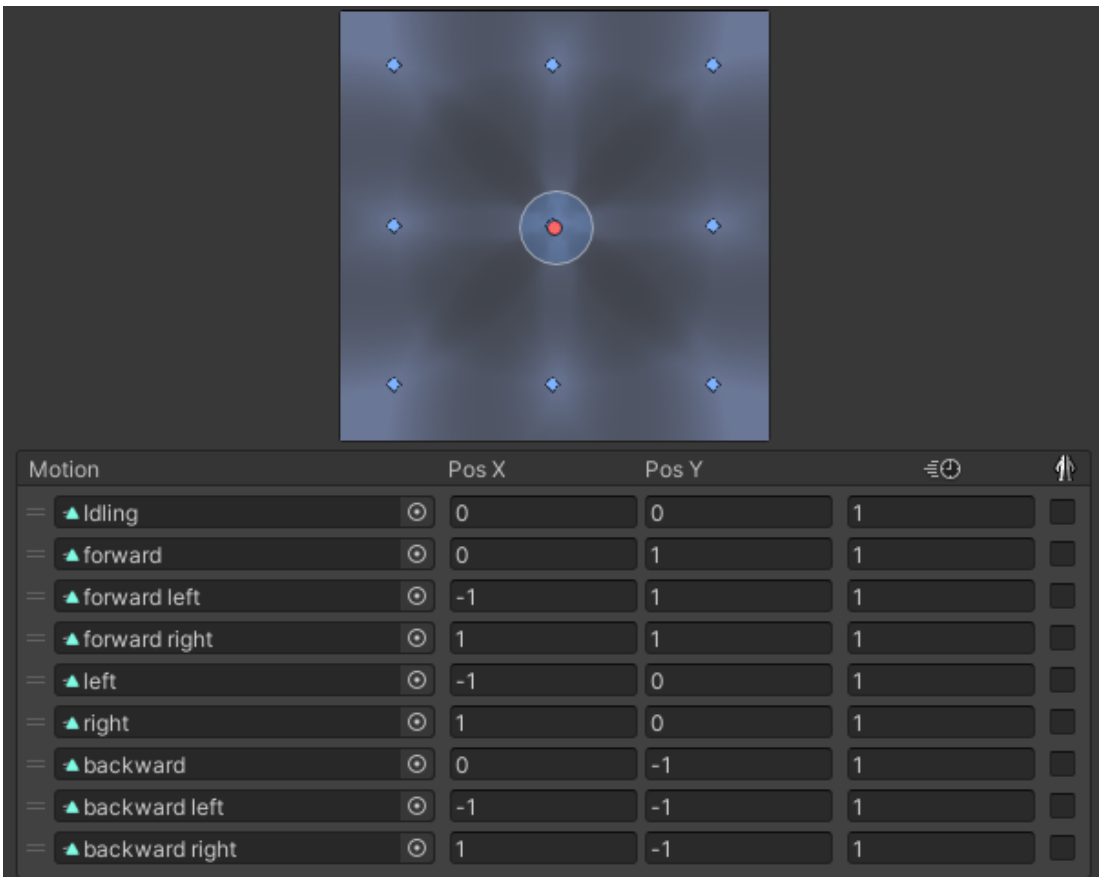
The blue box with the dots is just a 2-dimensional coordinate system with an X- and a Y-axis, both spanning from 1 to -1. Each of the blue dots represent one of the animation clips. The red dot indicates what is currently considered ‘active’, so by moving this dot around we’ll blend between the animations, as it moves closer or further away from the blue dots. By overlaying the thumbstick from earlier, it is easy to see how the position of the red dot can be fetched from where the thumbstick is moving.

By clicking on either each of the blue dots in the visual overview, or on the animation in the list, you’ll get a better idea of how this type of blending works, as the blue haze that you see in that window, will move around to show you what region the red dot would have to occupy to play that animation, with the strength of the blue color showing what degree the blending would be applied in:

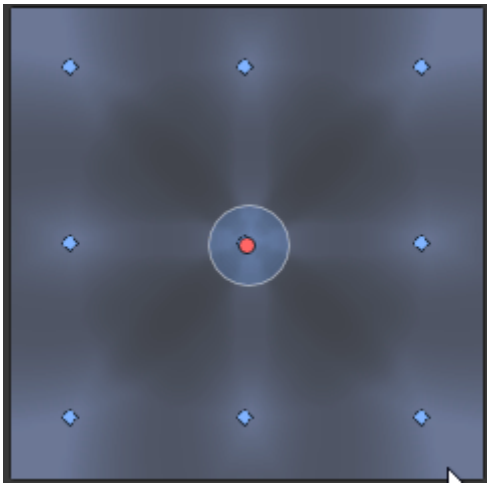


(PLAY GIF)

What’s useful about this visual representation of blending, becomes evident as we begin to position the dots corresponding to the animation clips. So ‘idle’ is right in the middle, ‘forward’ is above, ‘backward’ is below etc. For reference, this is a good way to set it up, but you can adjust the numbers or move the dots by dragging them with the mouse, if you want to change how each animation is blended in when moving:



Notice how the shape of the blue haze has changed in response to the distribution of the dots:



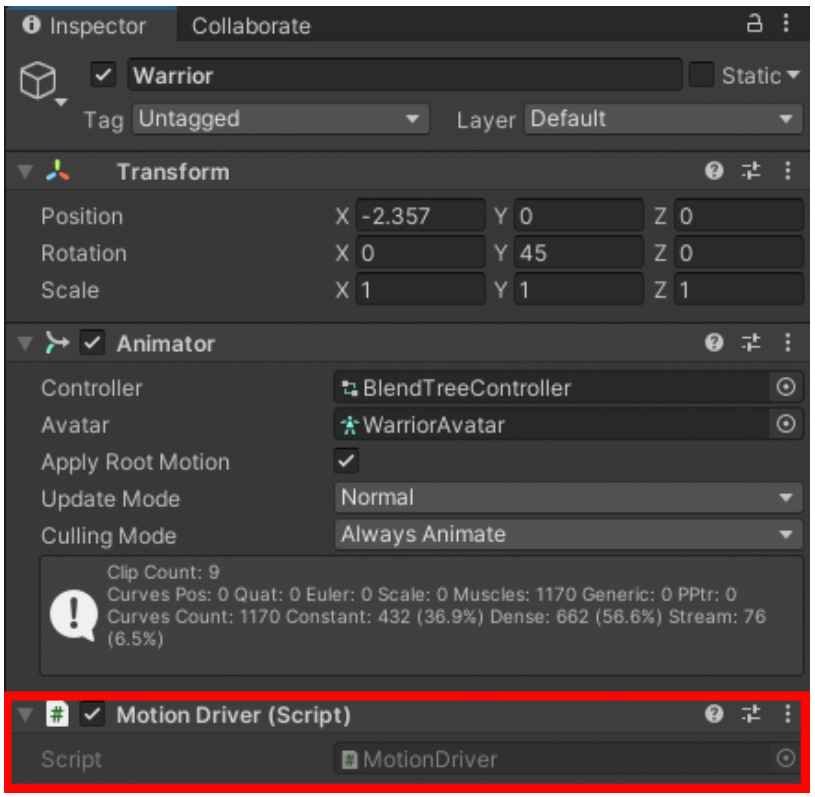
(PLAY GIF)

4.4.1 Driving blend trees:

Now that there's a blend tree with animations for multiple directions set up, we have to start moving the red dot around and actually play and blend between the animations. As mentioned earlier, we can easily do this by taking input from a thumbstick or a keyboard, but it requires a little bit of programming to access and combine these two things together.

Let's start by making a script called "AnimationDriver". I recommend that you normally choose something that is more appropriate for the context you are deploying it in (such as "PlayerMovement"), but I picked this one because it fits with the educational context that I am using it in. ***Just don't name anything in your native language, like how Ton Roosendall wrote his code in Dutch when he started making Blender.***

Since we're going to be controlling the Animator component of the character, it's fitting that we put this script on the object that has this component:

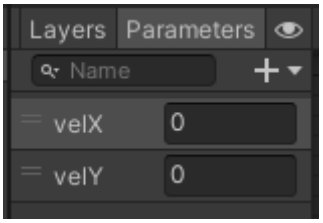


I'll paste the script here and explain each part, as it is very simple. **Be aware that more tweaking and additional logic is required to make it look and feel *good***, *this tutorial just focuses on basic implementation logic.*

```
using UnityEngine;

public class MotionDriver : MonoBehaviour
{
    Animator animator;

    // Start is called before the first frame update
    void Start()
    {
        animator = GetComponent<Animator>();
    }

    // Update is called once per frame
    void Update()
    {
        animator.SetFloat("velX",Input.GetAxis("Horizontal"), 0.1f,
Time.deltaTime);
        animator.SetFloat("velY",Input.GetAxis("Vertical"), 0.1f,
Time.deltaTime);
    }
}
```

After we've initialized the animator variable, the Update function is where the logic happens. Here the **animator** variable has the **.SetFloat** member function. This is used to control the blend tree parameters that we declared earlier - *velX* and *velY*. Here's how this logic works:

Name - Must be identical to the names we've used in the Animator:

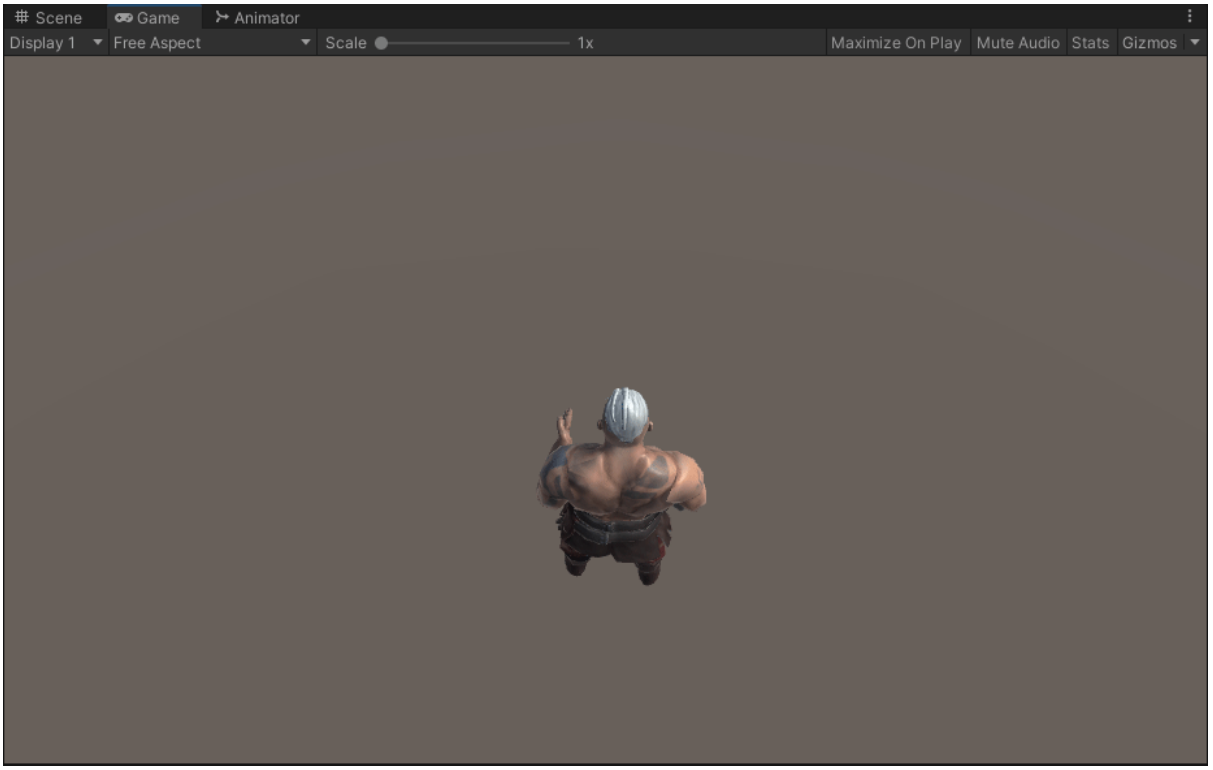


Value - Since we're using the velX and velY parameters to move the red dot around on the blend tree, we're just getting the location of the thumbstick (or equivalent from keyboard) to determine the coordinates. *Unity has predefined definitions for "Vertical" and "Horizontal" when using Input.GetAxis, but I recommend playing around with Unity's Input system if you want to do stuff like letting players rebind keys.*

Damper - This determines how long it takes to blend between animations.

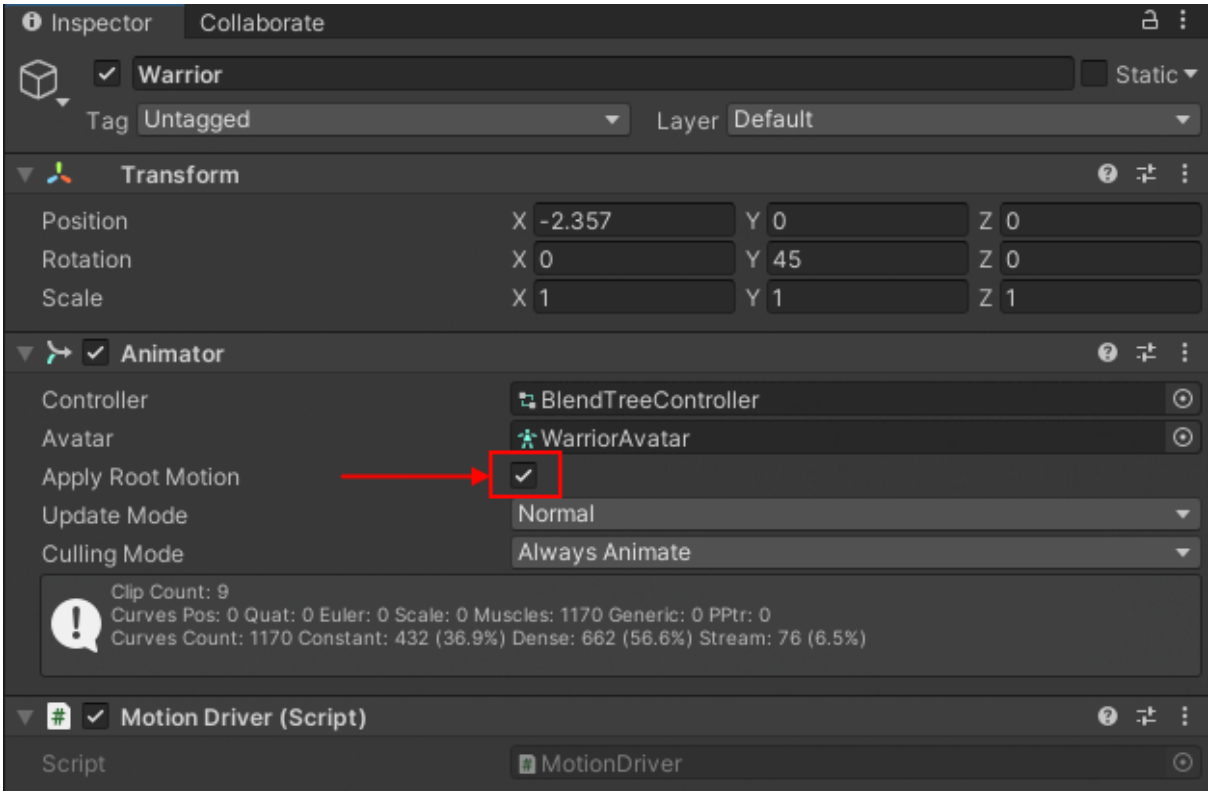
Damper delta time - This parameter simply considers how much time has passed in the system and acts as a multiplier for the Damper value. So if any lag occurs, it will not affect the speed at which the animations are blended.

If we now run the game and try moving the character with either WASD, the arrow keys or a controller, you'll get something like this:



([PLAY GIF])

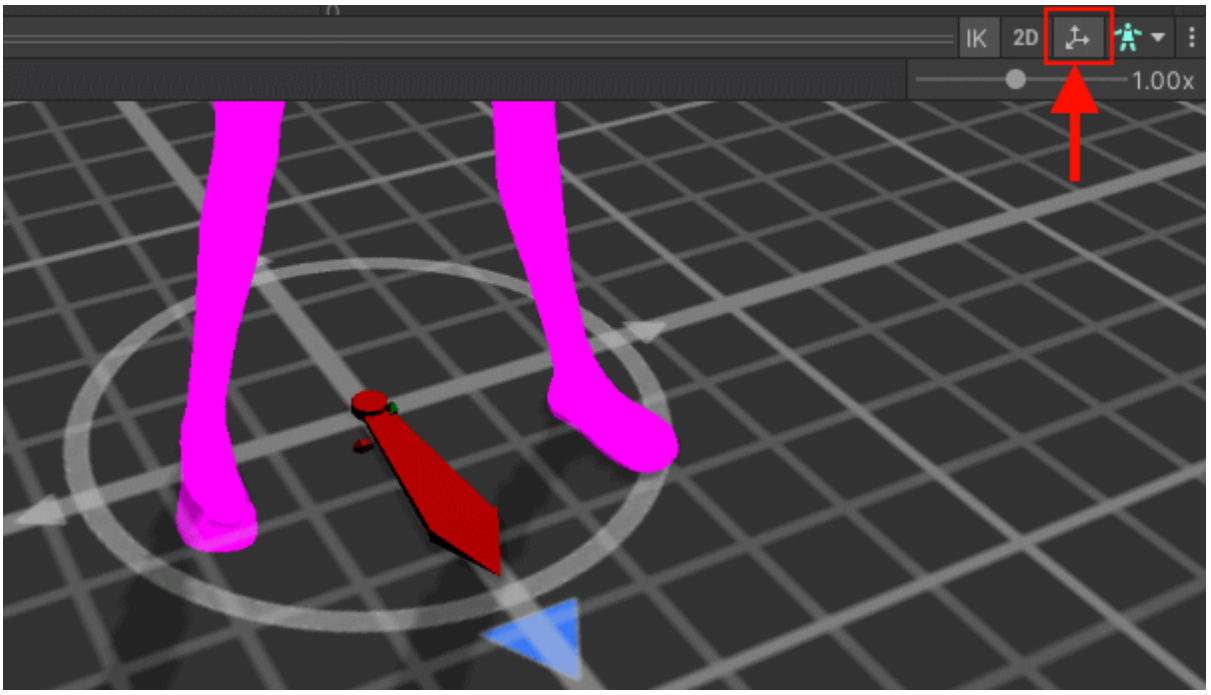
If your character doesn't move, select your character and make sure that **Apply Root Motion** is set to true on the Animator component:



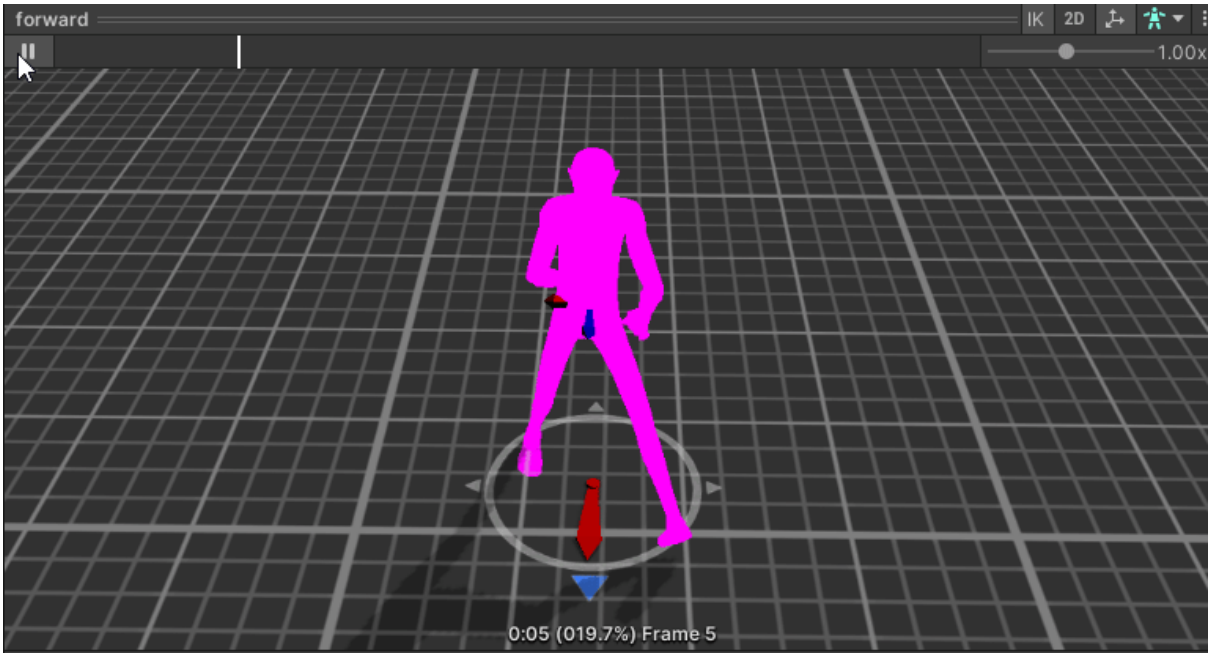
4.4.2 (BONUS) Root transform corrections:

As you may have noticed from the gif above, the directions are kind of weird. Odds are if you're doing mocap, the resulting animations don't have pixel perfect directional movements, which you'll have to make adjustments to. And since we've already touched down on root transform corrections when we were importing our fbx-files, we might as well round this chapter off by making use of that.

If we look at the Animation preview in an fbx file, you'll notice that there's a blue and red arrow (make sure "Display avatar's pivot and mass center" is toggled on):

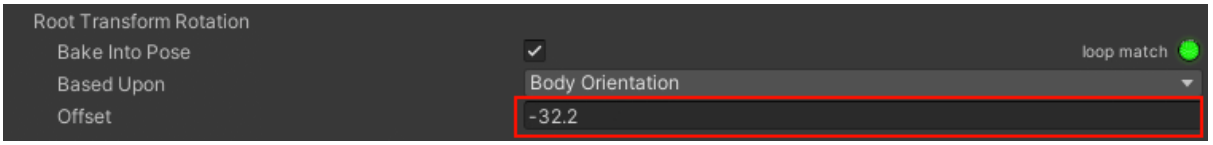


The blue arrow represents the root rotation, whereas the red arrow is an estimated guess by the animation system, as to where your character is facing (not sure exactly how this is calculated, maybe based on the head bone rotation). But if you are using the Mixamo walk animations that I previously suggested, you'll notice that the root "forward" is not actually the direction that the character is moving when pressing forward, but rather in the direction that they are aiming their invisible gun:

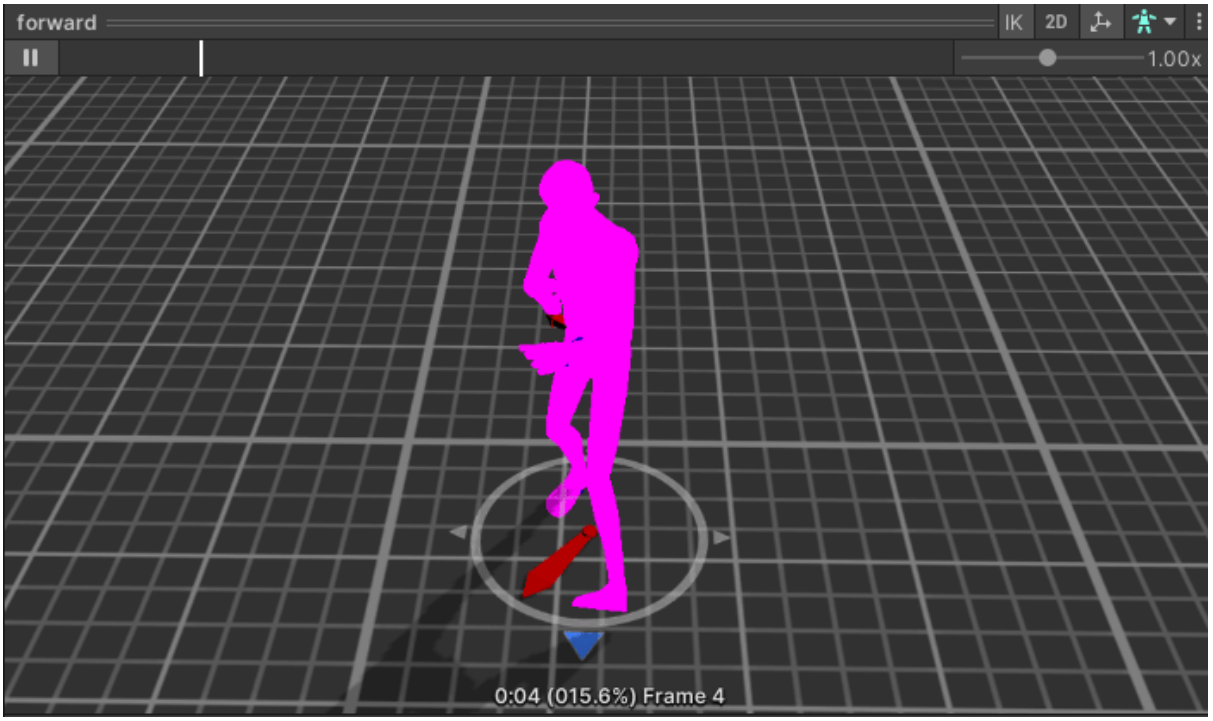


([PLAY GIF](#))

That means when I am pressing forward in the game with our current setup, the viking character actually walks sideways. In order to fix this issue, go to the Root Transform Rotation setting above the preview, then change the value. I recommend just hovering with the mouse, the click and drag right/left to increase/decrease the value, until you find a good rotation:



Use the grid on the ground to determine the direction of the character's motion. As you may notice when doing this, the blue arrow will still be pointing forward, because the object itself is not rotated by this approach - so a script referring to the forward direction would get something pointing in that direction still. But the red arrow will change as the animation is rotated. It can be a little finicky to get it to walk straight, but your character does not have to walk in a perfect line, as long as it feels good. Mine ended up looking like this:



(PLAY GIF)

If anything looks or feels off in terms of direction, I recommend that you preview your animations like this and make adjustments. It works for both when baking rotations and not baking, so regardless of whether you are driving your motion with scripts or animations, this process is applicable.

4.5 Face Capture - Implementing blendshape animations:

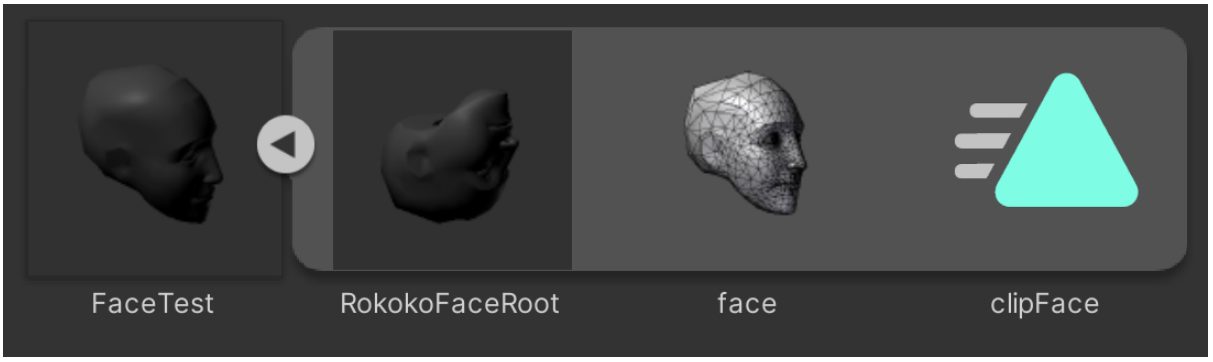
When working with blendshape animations such as Face Capture clips from Rokoko Studio, there's a few things we have to consider, which may involve using Blender (or whichever program you prefer) to change a name or two, so Unity is able to couple things together. **Be aware that this process only works for characters that share the same blendshapes and blendshape names as those that are defined in Apple's ARKit, as this is the basis for our Face Capture app.**

I am going to do an example case using a Newton face capture export, which I will be putting onto the viking character. There are two things that we need to address with our character in Unity before the blendshape animations will play:

1. Mesh name
2. Object hierarchy

To illustrate the problem in general, I'll begin by importing a face capture recording from Rokoko Studio and take a look at the resulting fbx file:

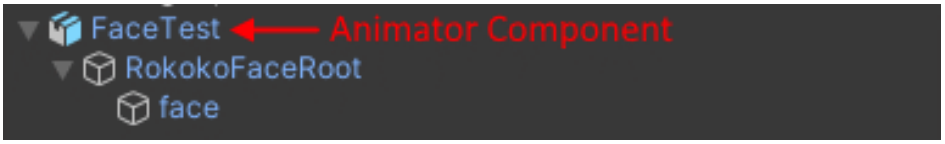
In the Project view:



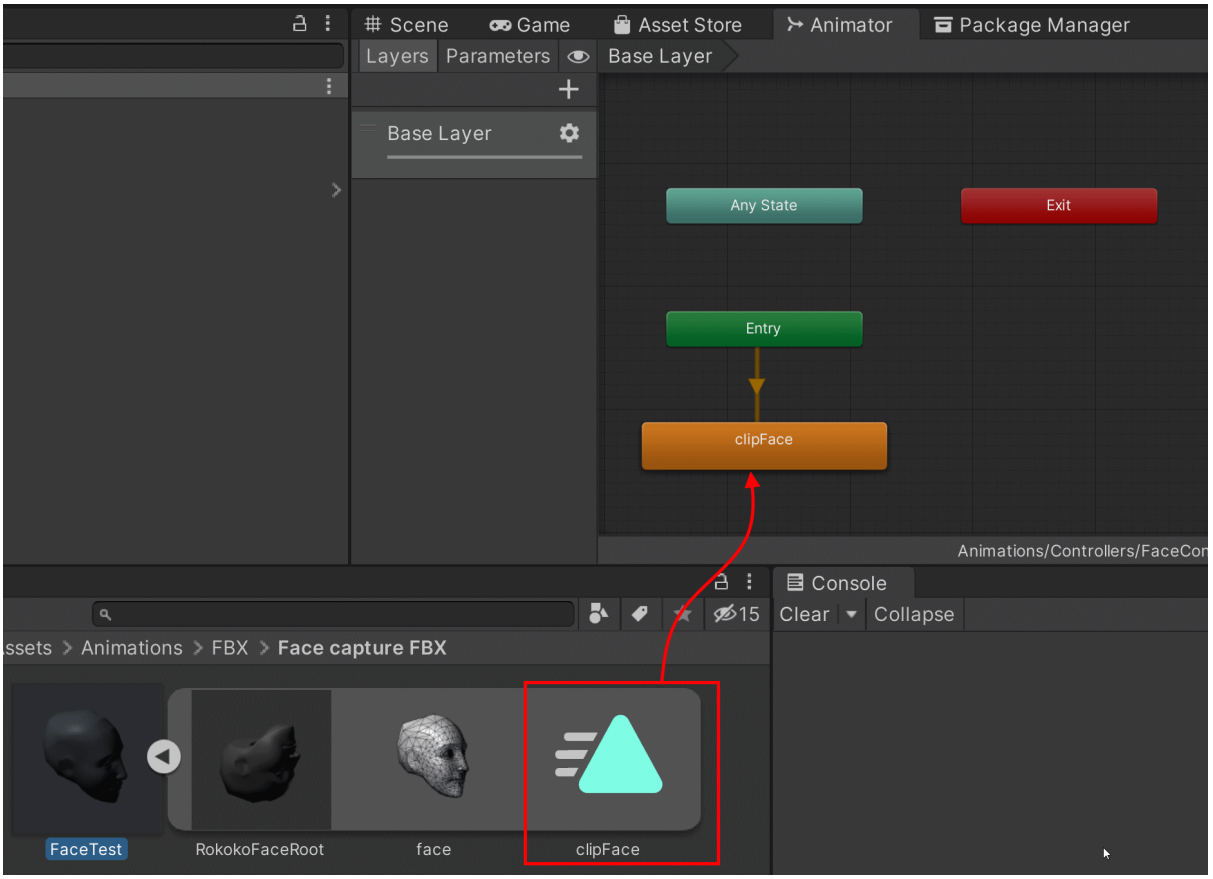
This view shows three things - the object (upside-down face), the mesh (untextured wireframe face) and the animation clip.

In the scene view:

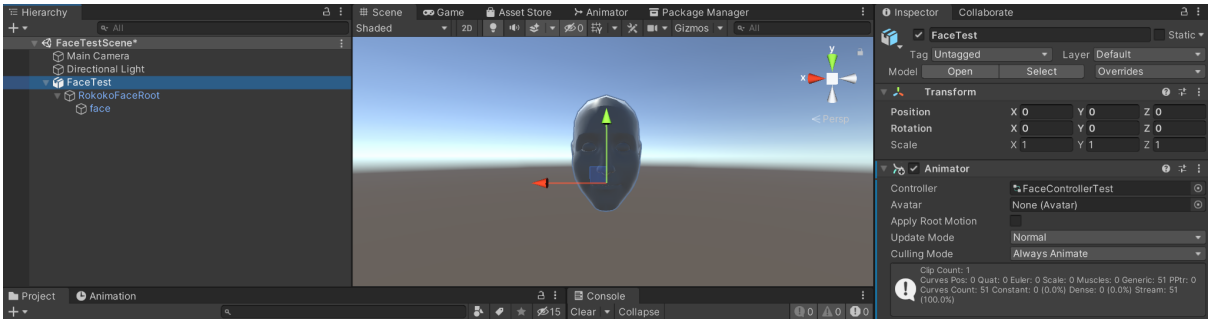
This is what the face capture fbx file looks like when you open it up in the scene view. I have added the red text to illustrate where in the object hierarchy the mesh is actually sitting, as this is very important. In order to get the clip to actually play on on this object, you will have to put your Animator component here:



For a quick test that this works as intended, I recommend creating a new animation controller and just having it play the face capture by default. I've made one called FaceTestController and just dropped the FaceControllerTest fbx in to be the first thing that plays:



Go back to the FaceTest object in the scene and select this controller to be used for the Animator that sits on the root of that object:



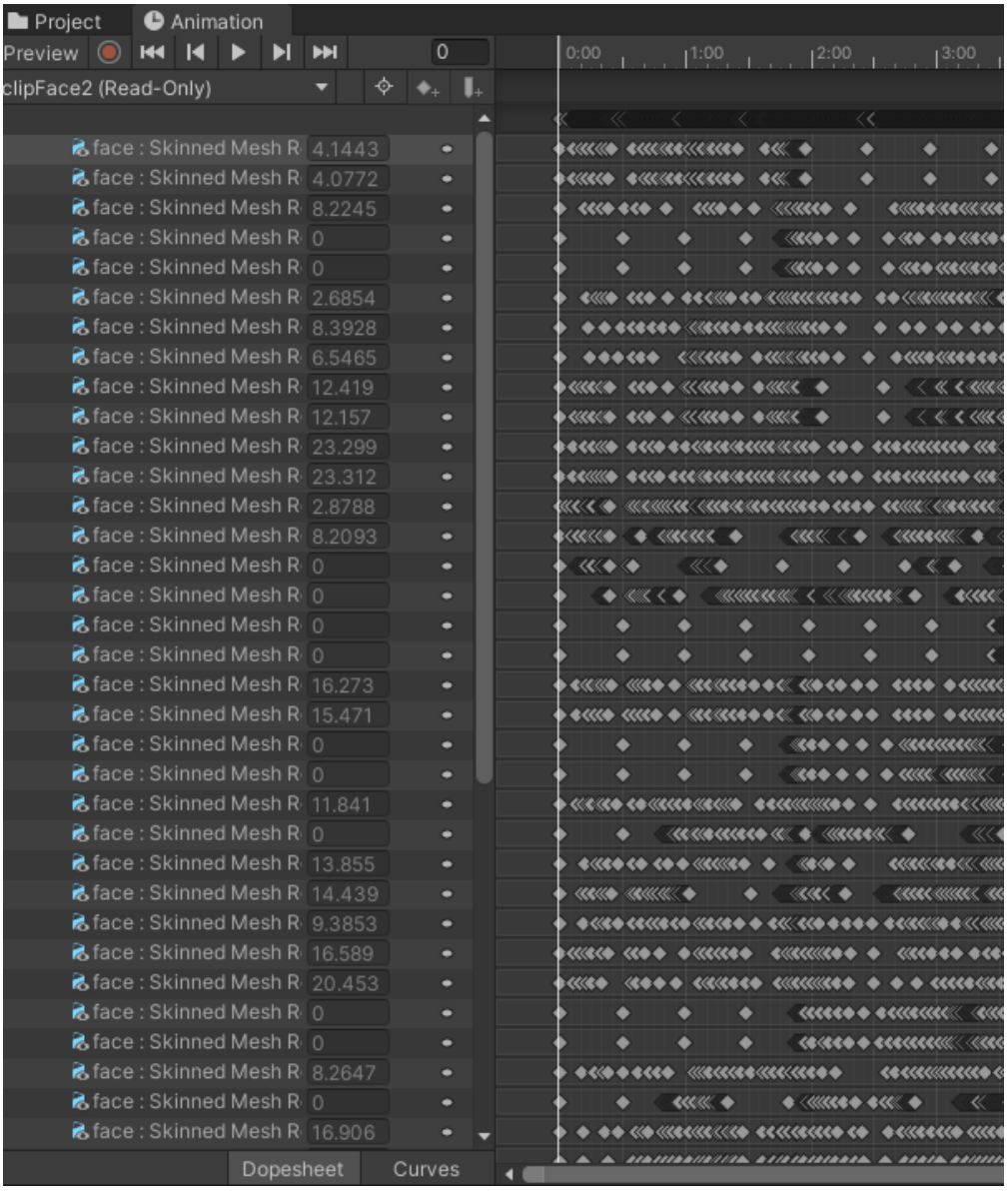
And you should get see it moving like this:

(PLAY GIF)

Okay - so we talked about object hierarchy and mesh name and confirmed that the animation clip works if we just slap it on the default Newton face. But we have yet to explore *why* these things are important and how we can make use of that knowledge to make the clips work on our own characters.

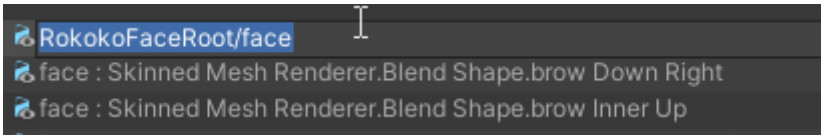
The key to understanding this lies in the animation clip itself.

So let's open that up and look at what's actually happening. Double click on the animation clip in the fbx file and it'll show you a window with all of the keyframes like this (if it doesn't show anything, you can ask it to show read-only data on the left side of the window):



What you see on the left side are each of the blendshapes that this clip is animating - an on the right side is the timeline and keyframes for each of these blendshapes. Let's take a closer look at just one of the shapes:

Notice how there's a little blue icon to the left? If you click that, you'll see something really interesting - **the hierarchy which this animation clip is using:**



What this means is that **the animation clip, playing *from the location of the Animator component*, is looking through the child objects for this hierarchy.** This matches the hierarchy that we see in the face capture fbx files when we look at them in the scene view and it is the reason why we have to use the same names in our objects.

In other words - if you have a character and you put an animator on it and expect it to be working, you will have to match this ‘environment’ of naming and hierarchies in your object.

4.5.1 Renaming the mesh

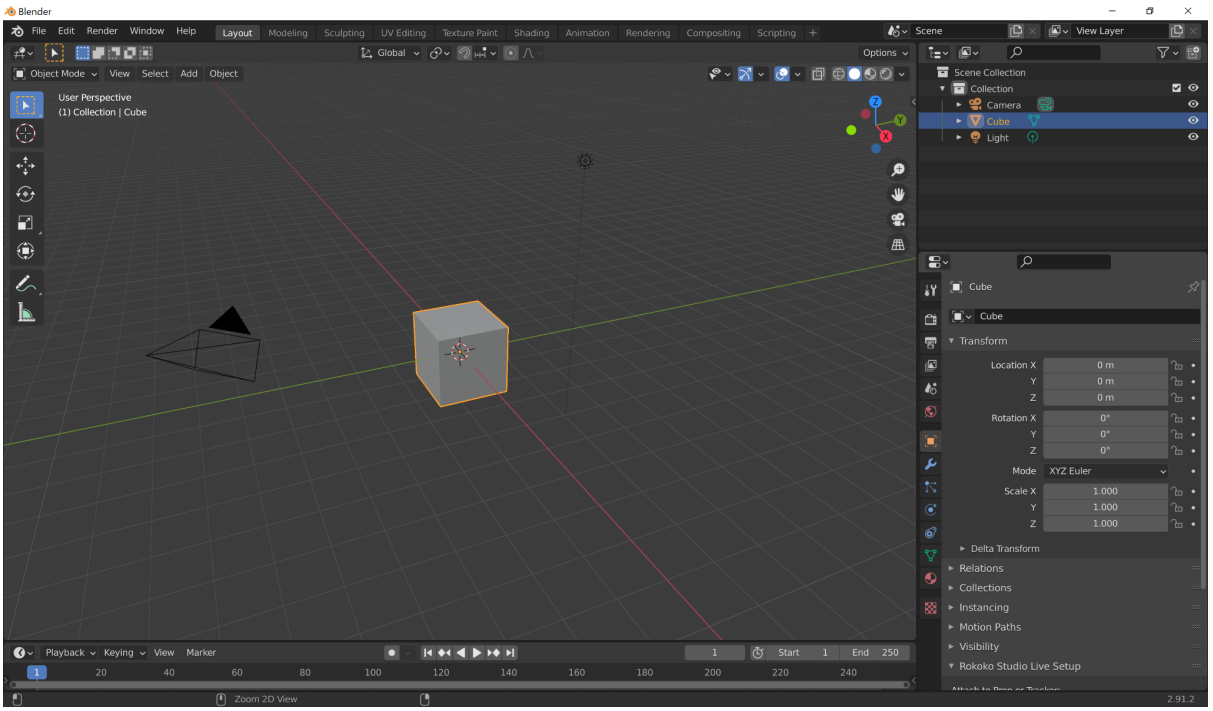
Unfortunately, Unity does not have an easy way to do this in-engine (though plugins like Skinn Pro could be used - I’d rather show you how to do it for free).

Fortunately, Blender is super easy to use for this exact purpose and it is free forever, so I recommend that you go and grab it here:

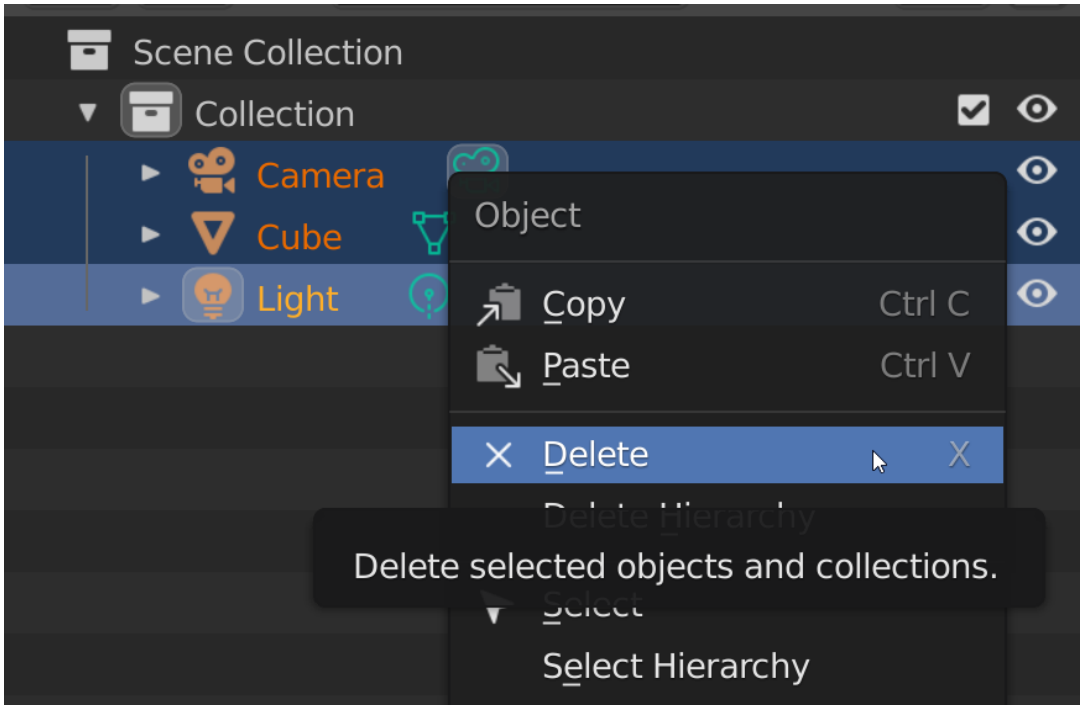
- https://www.blender.org/

You can of course do this in whatever program you prefer, as long as it is able to unpack, edit and export fbx files, but I’ll do this walkthrough in Blender because I think it is an awesome program.

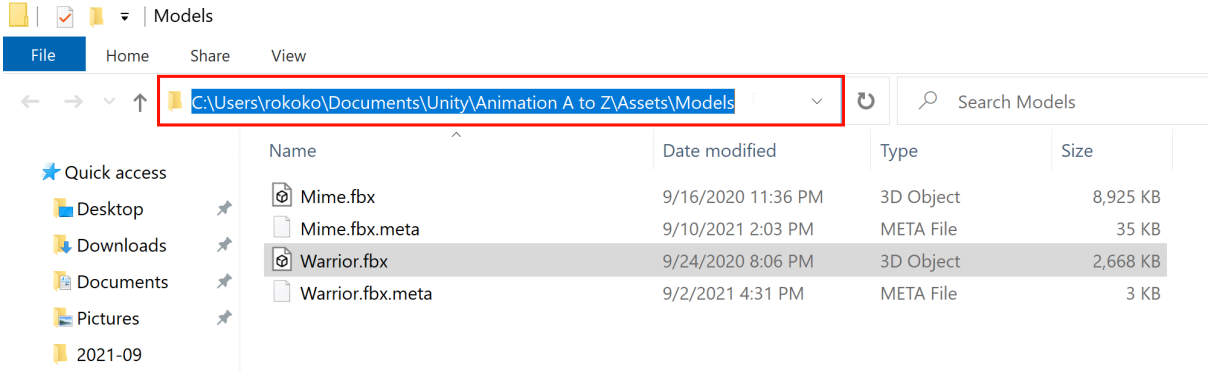
Once you open up Blender and click past the splash screen, you’ll be greeted with a scene looking like this, with the (in)famous default cube in all its glory:



Before we start importing anything to be edited, look at the upper left where you’ll see something resembling the Hierarchy view from Unity, listing a Camera, Cube and Light object. All three of these should be deleted, because anything we export will contain every element in the scene, and we don’t want to glue any lights or cameras to our characters, we just want to rename the meshes:

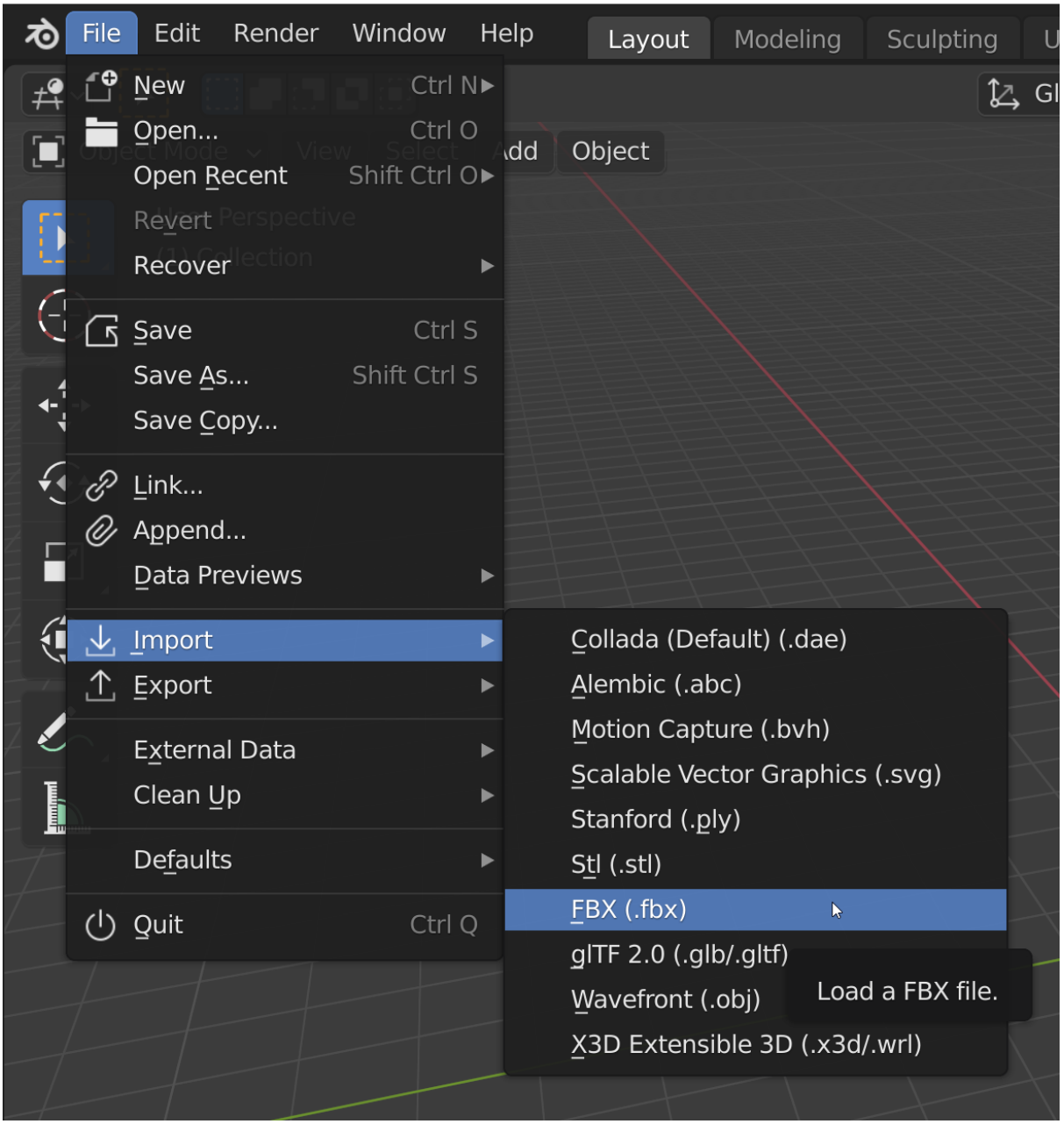


Now with an empty scene, we can import the Viking character. The easiest way to do this is to find the fbx asset in Unity -> Right click -> Show in Explorer, then copy the folder location:

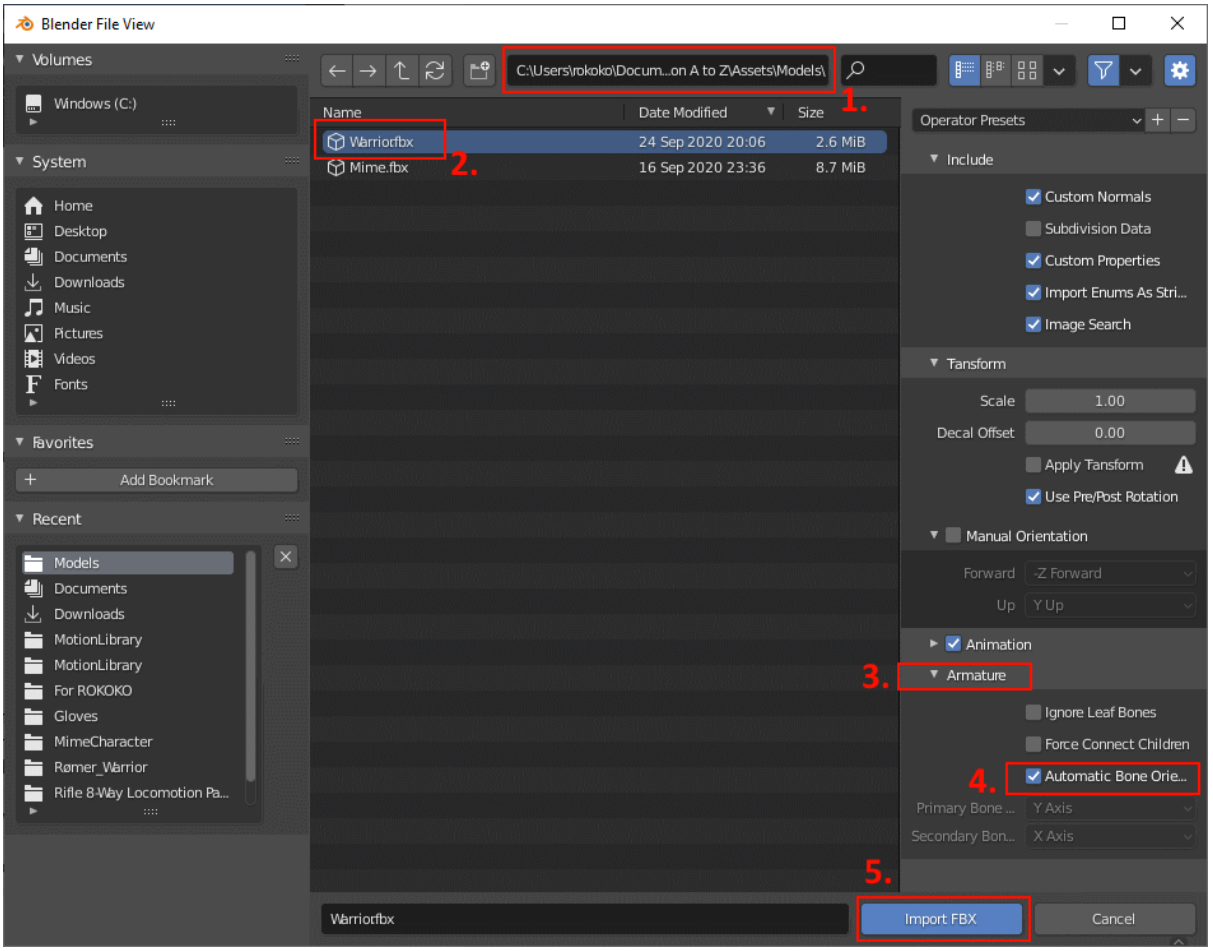


Go back to Blender and select File -> Import -> FBX from the upper left corner:

Blender



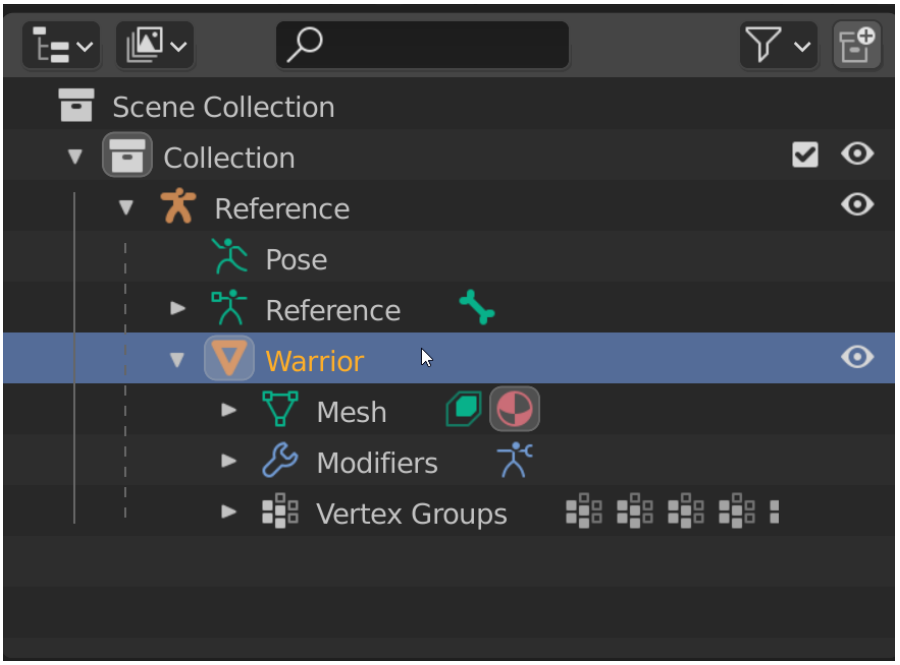
This will show you Blender's file browser, where you can paste the folder location and select the fbx-file that you wish to modify. Here's all the clicks you have to do:



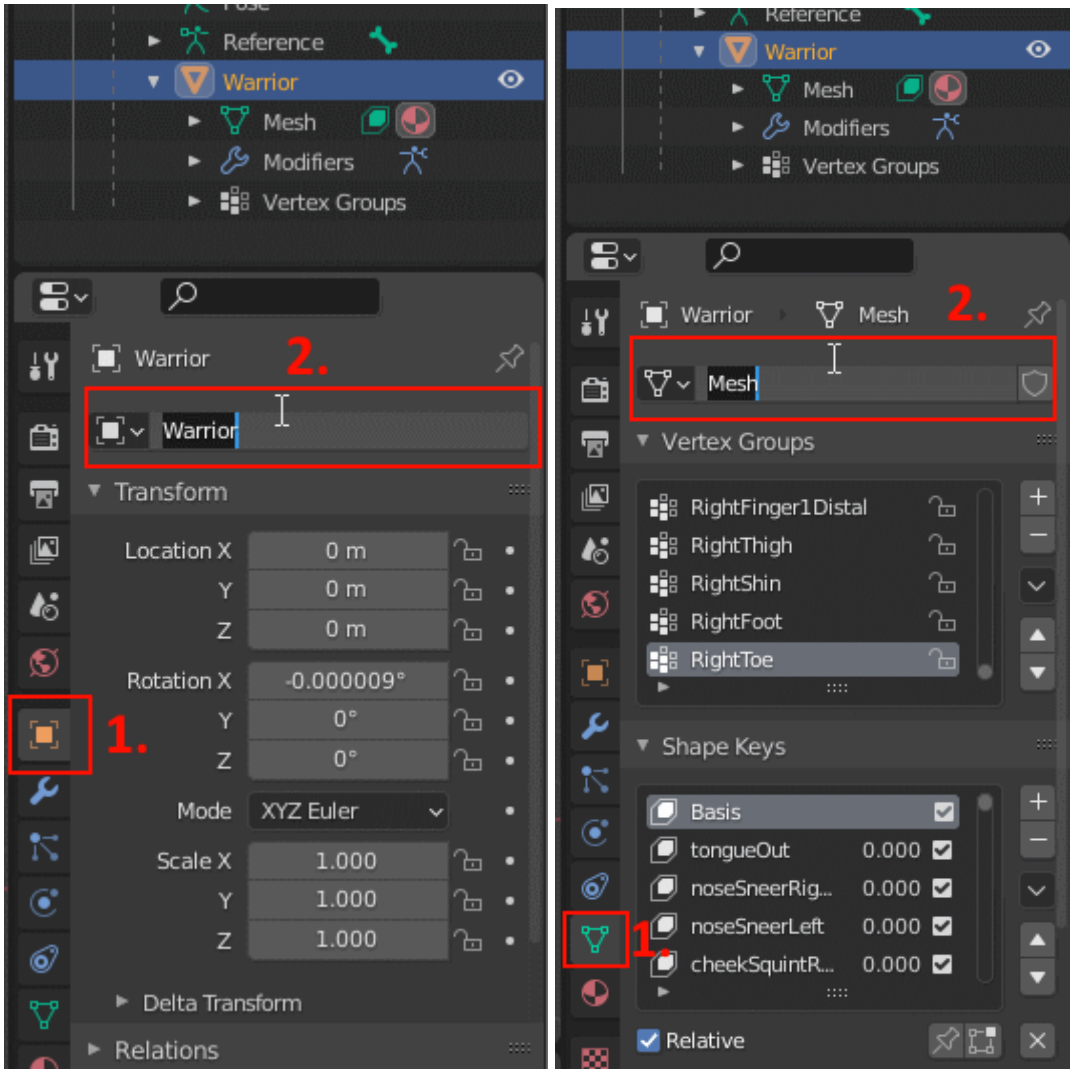
1. Paste the folder location and press enter
2. Select the fbx file you wish to modify
3. Expand the Armature import options
4. Select Automatic Bone Orientation
5. Click Import FBX

If everything went well, your imported character should look something like this:

If we look at the scene collection in the upper right corner again we'll see that the character is now showing as an expandable object. If you click the arrow to expand the view, keep an eye out for an orange triangle-symbol, which is where the mesh is located. On the Viking character, it is located here:

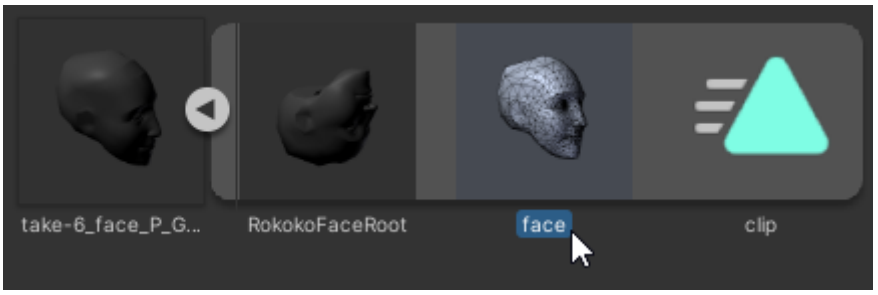


To change the name of the mesh, we will have to change the name in two different places, before finally exporting the character again. If we click this object and look below the collection view, there's an assortment of tabs to choose from. We're looking for the ones called **Object Properties** and **Object Data Properties**, each of which has a place where the name can be changed:

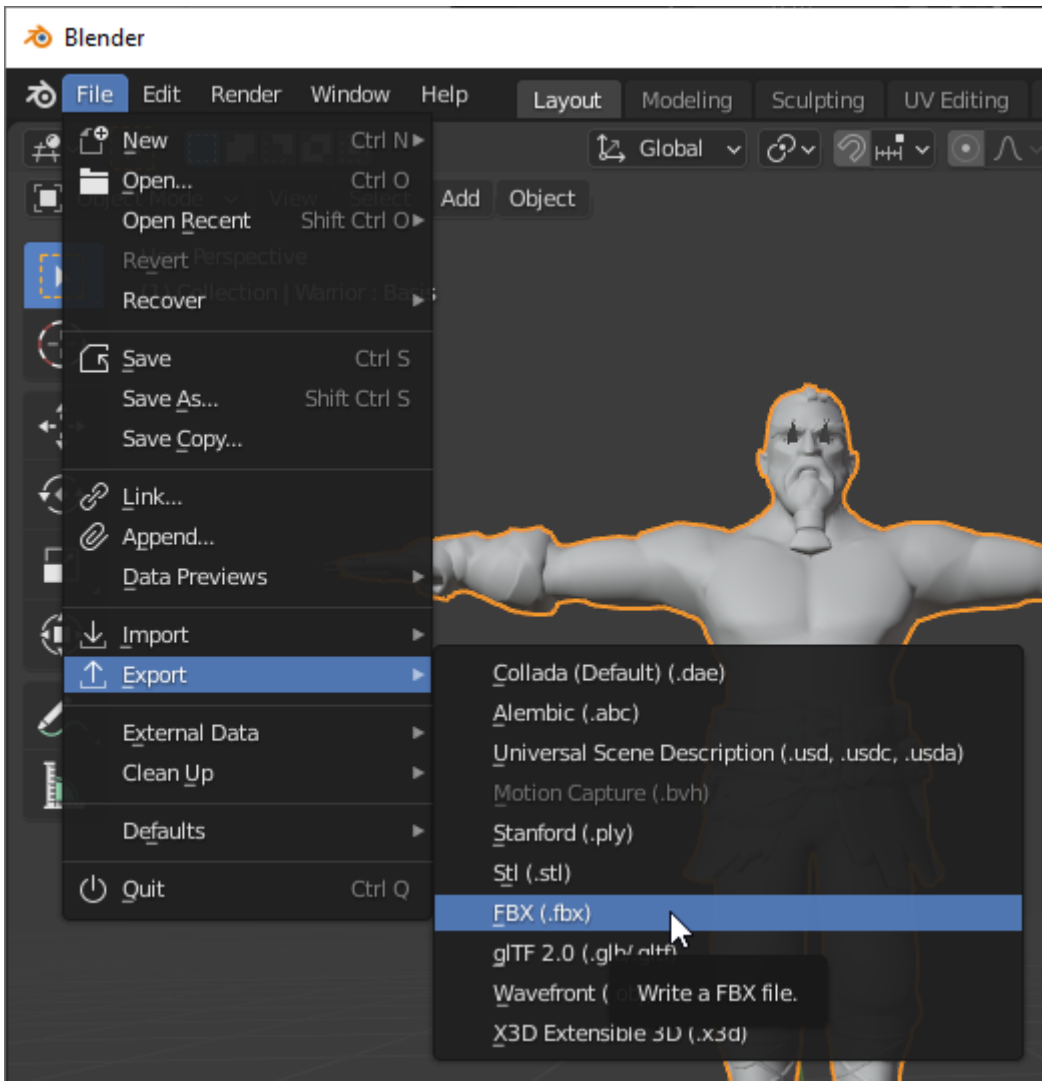


Change these names to **"face"**

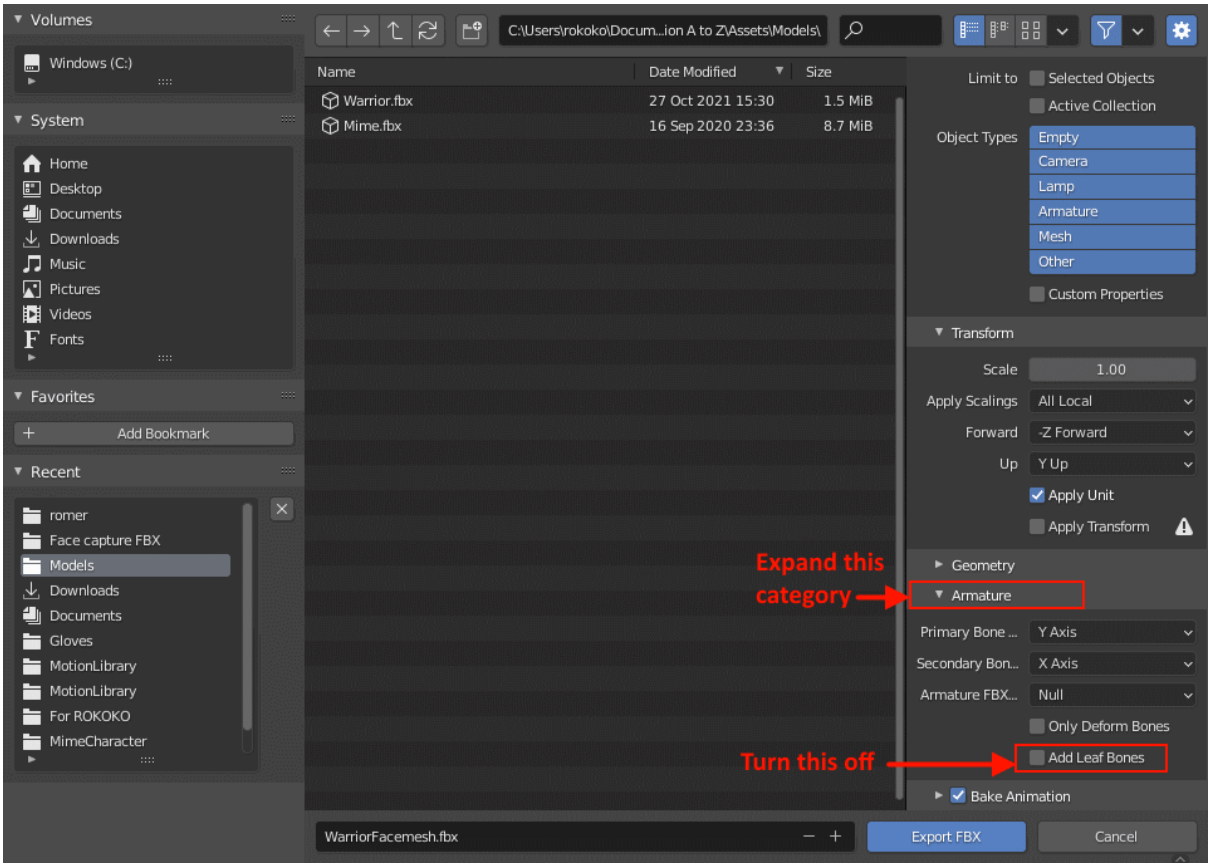
It may be enough to just change the name in the Object Data Properties, but it is good practice to match the names and I am not 100% sure which of these that Unity uses as a reference for naming, so I recommend always changing both. In this case **I am changing the name to "face"**, as that is the same mesh name that the Face Capture export uses. If you are working with face animation from a different source and you aren't sure what the name is, the easiest way to see the mesh name is just to expand the fbx file inside of Unity and look for it there (it'll always show as untextured in wireframe):



Once you have changed the names, export the character from Blender again by going to File -> Export -> FBX:



If you paste in the folder directory from before, you can overwrite the existing character file with the new naming. I recommend that you save the changed version of the model with a different name, but just in case you don't, you might want to recreate your character prefab (you'll see what happens, lol). *Also - please keep in mind that when you export, Blender likes adding leaf bones to your skeleton by default. If you don't know what this does, I recommend turning the setting off:*



If you forget to do this, you can ignore leaf bones when importing in Blender, to remove them again.

If you save the file to your Unity directory, it'll automatically re-import the fbx file when you click into Unity. Now don't be scared, but if you saved directly on top of the existing fbx, you'll very likely be greeted with this something looking like this monstrosity:

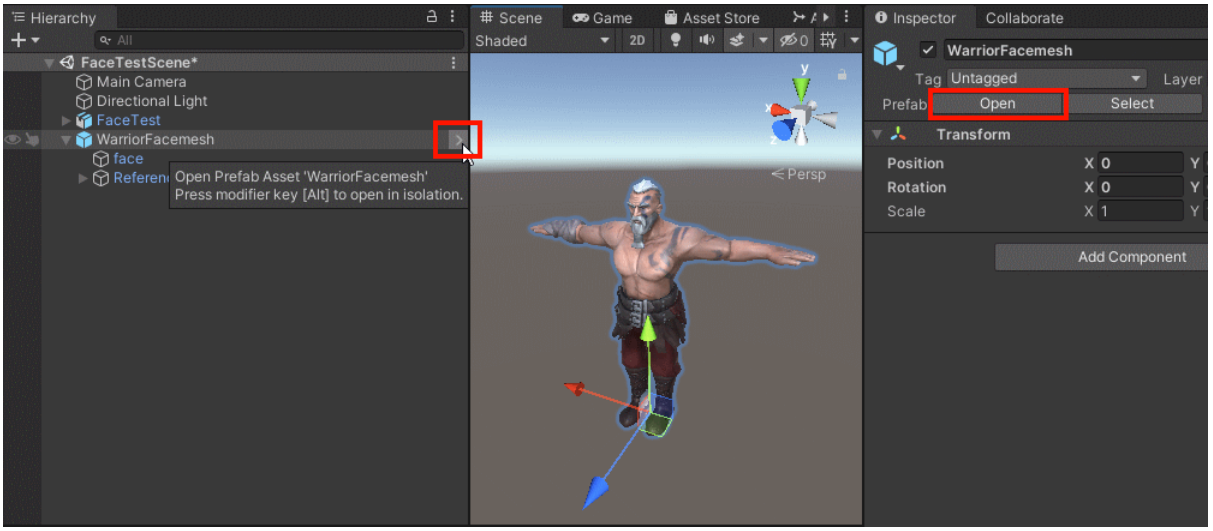
I am not exactly sure what happens, but going outside of Unity and messing around with the fbx file directly, has likely moved some data into the wrong places. My suggestion to fix this is simply to:

1. Recreate the humanoid rig for that fbx file, which you can do by setting it back to Generic, hitting Apply and then setting it to humanoid and hitting Apply again.
2. *Recreate the character prefab by dragging the fbx file back into the scene and reapplying the scripts, then save it again as a new prefab by dragging the fbx into the*.

The resemblance is uncanny.

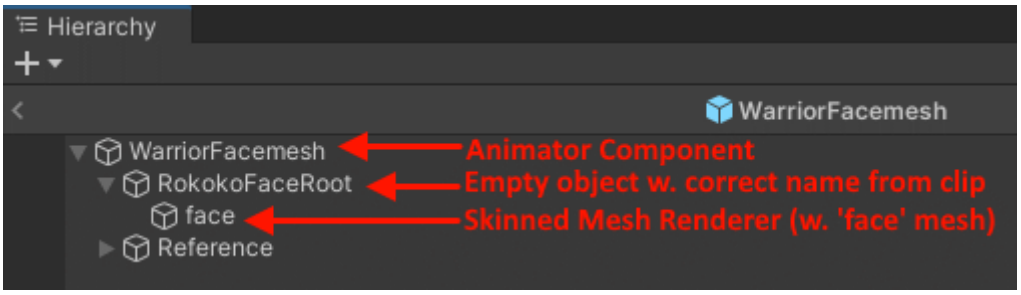
The reason I am not renaming the Face Capture fbx instead of the character fbx, is simply because that means I would have to rename each of these clips with the character name, instead of just renaming one character to the face name.

But now that the mesh names match, the last step is to modify the prefab where we have reorganized the objects to match the expected hierarchy. To do this, open the prefab by double-clicking on it in the project view, or through the hierarchy or inspector by clicking either the little arrow next to the prefab, or the “Open” button:

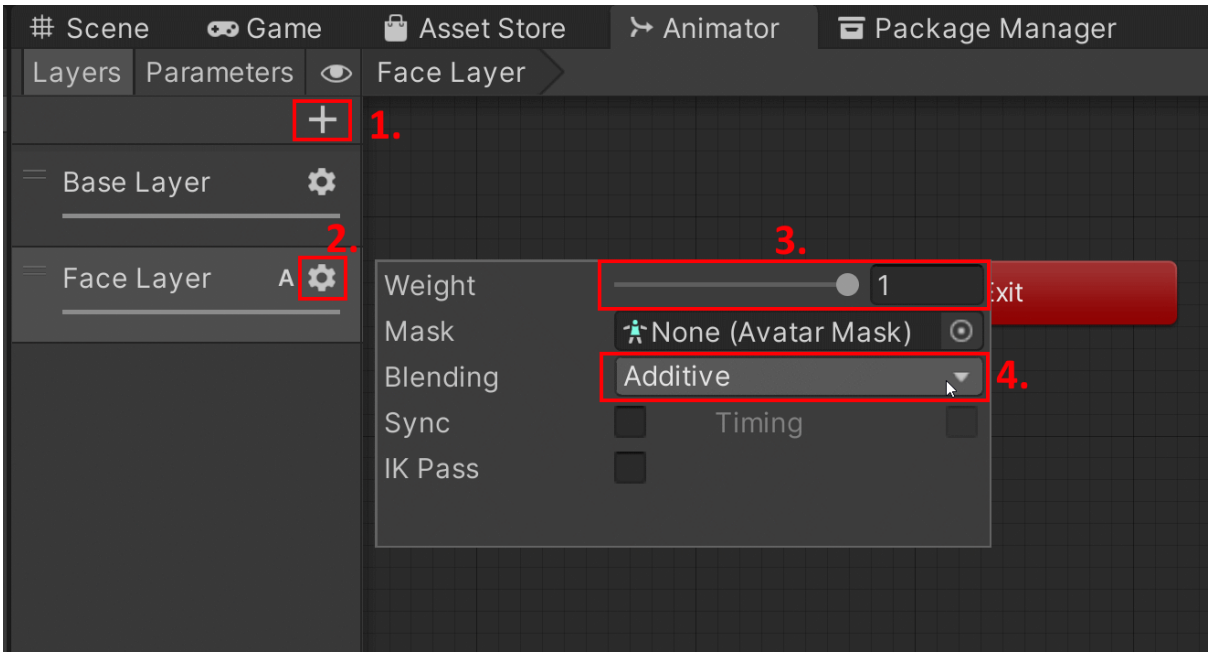


Again, here is what my character looks like in the hierarchy view at the moment:

We know that the animation clip looks for the hierarchy "**RokokoFaceRoot/face**", so I am going to create an object with that name in the prefab, childed to the object that has the animator component on it, then child the "face" object to that, so we'll end up with a prefab that looks like this:



Finally, we can expand our animation controller to include a layer that plays the face animations, on top of the animations that control the body:



Open up your animation controller and:

1. Add the new layer
2. Open layer settings
3. Make sure that weight is set to 1 (plays blendshapes to their full weight)
4. *Set the blending mode to **Additive***

I renamed the layer to Face Layer as well, just to keep things tidy. Now if you've done everything correctly, adding animation clips to this layer should animate the face of your character like this (Newton side-by-side for comparison):

(PLAY GIF)

4.6 Playing multiple skeletal animations simultaneously:

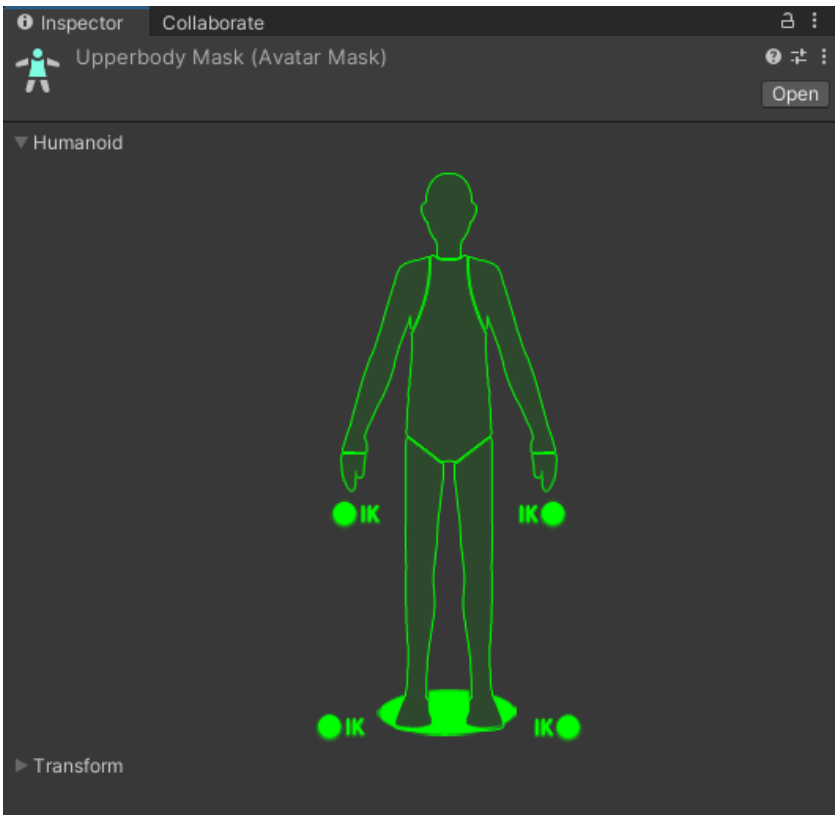
We've briefly touched on this with the blendshape animations, but it is possible to create animation layers that exclusively target certain bones of a character as well. A classic example is running and swinging a weapon at the same time. While we've been animating the entire body of the character with one animation so far, we want to selectively take control over the upper body with another animation, when the related action is carried out.

The setup for this is fairly simple - it requires us to create a separate animation layer, just as we did for the face capture, but this time creating a "mask" that specifies what bones the animation is allowed to effect. In the project, **right-click -> Create -> Avatar Mask.**

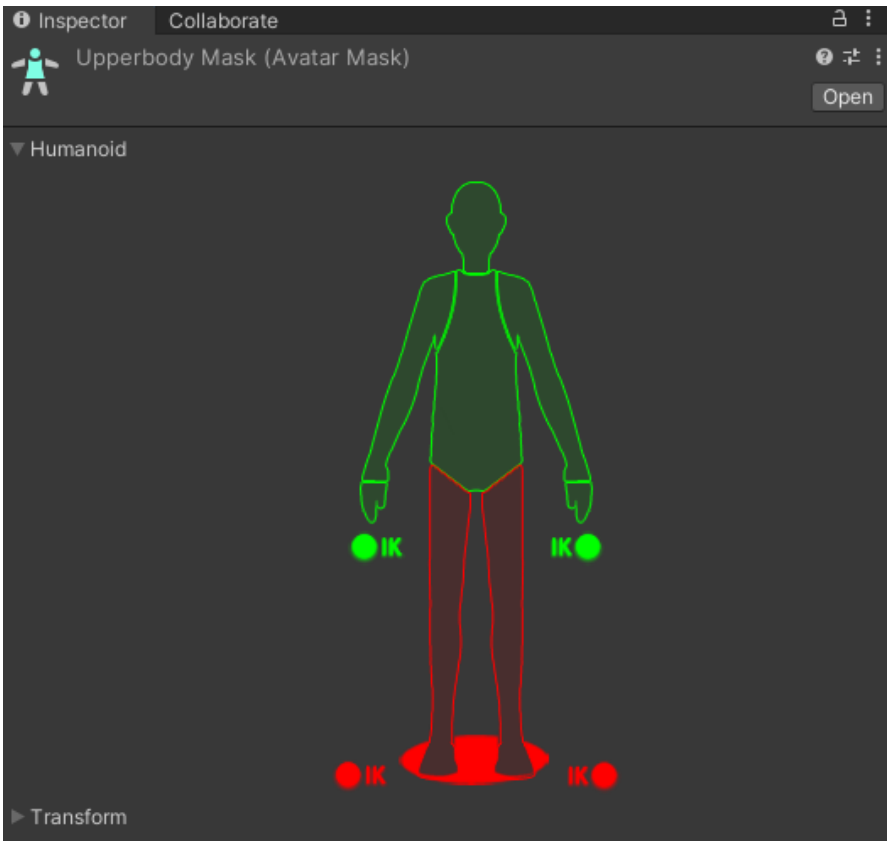
When you select the avatar mask and view it in the inspector, you'll see that there are two sections:

- **Humanoid**
- **Transform**

If we expand the first section, you'll see a view like this:

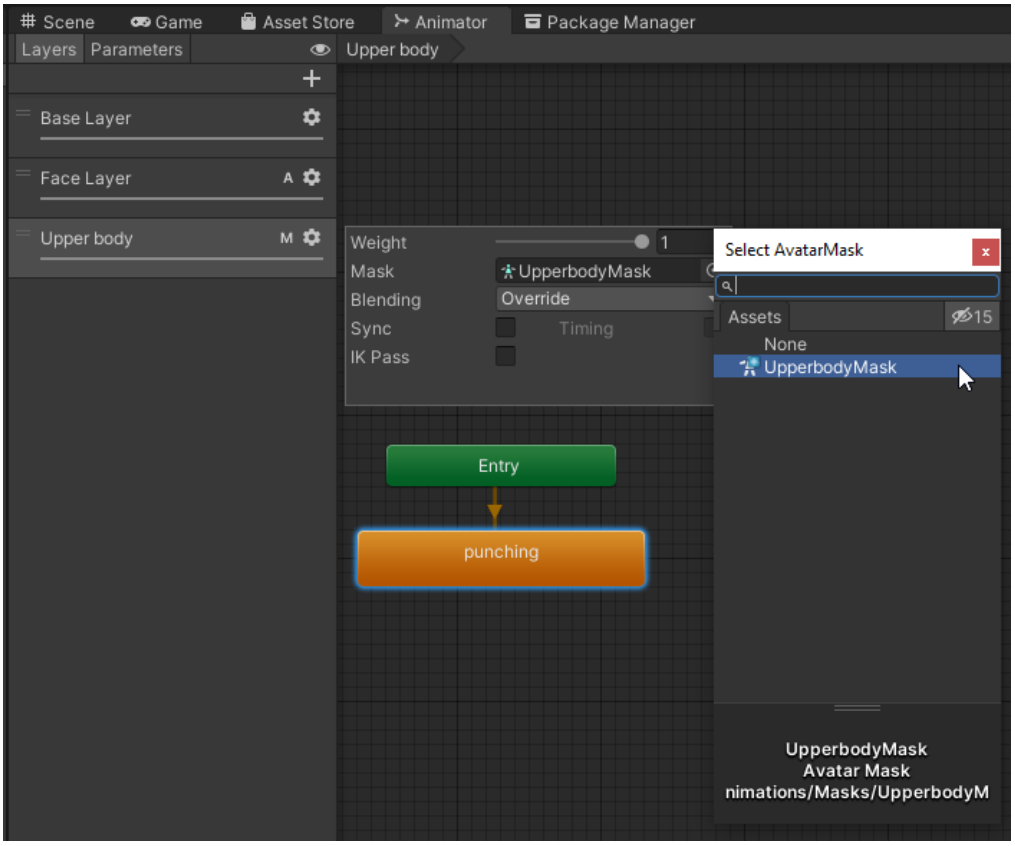


This is a clickable overview - anything shown in green will be affected by an animation that uses this mask. For this example, I've grabbed a punching animation off of Mixamo where the character swings their arms for a punch, while standing still. We do not want this to affect the legs or the root motion, so we deselect those parts and end up with a view that looks like this instead:



Note that I've also deselected the IK targets (the red dots saying "IK") even though we aren't using them at the moment. This is good practice, just in case you do decide to do anything with them later, which is very often the case when working with game logic.

Now to test that this works, I've simply set the punching animation to loop and created a new animation layer. I've given it a weight of 1, set it to override and made sure to select the mask that we just made:



While we do see a small problem in the final result, caused by the weird “aim walking” and the punching animations wanting to go in different directions, it is still quite evident that this works:

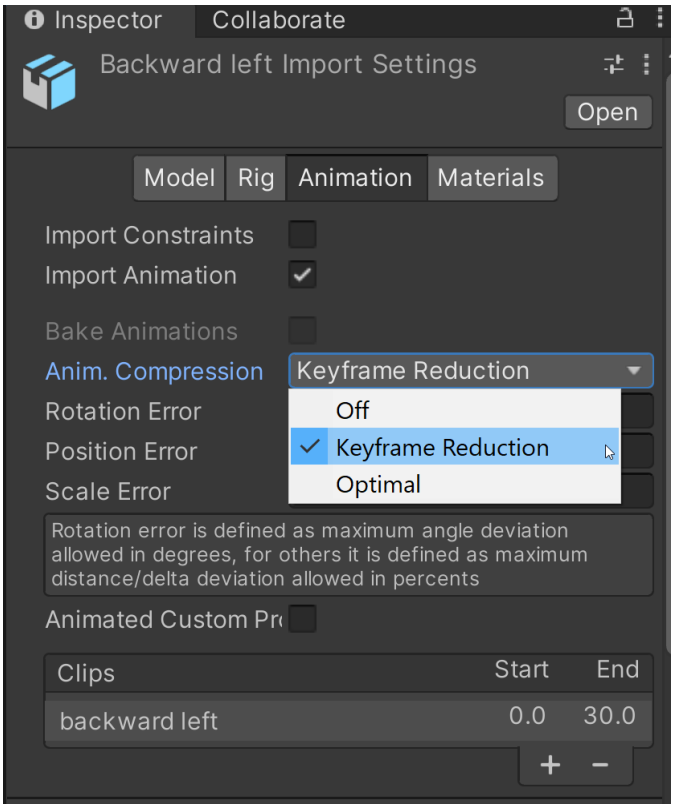
(PLAY GIF)
He’s a little shy, but he’ll punch your lights out.

4.7 Mocap cleanup (Blender, mostly):

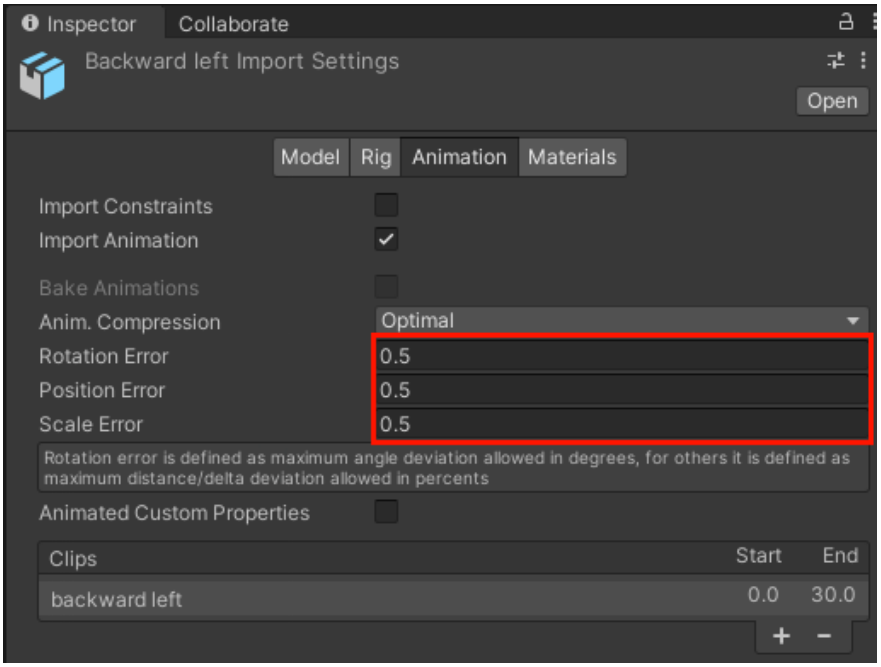
When working with mocap, you may have to do some cleanup to get your desired results. A common problem is having jitter in your motions, which Unity can compensate for to some degree, but you may want to do a more thorough cleanup of in Blender.

4.7.1 Removing jitter:

Unity's fbx tools do not have a dedicated "remove jitter" feature, but it does have animation compression, which inadvertently does reduce jitter to some degree. The way this works is that Unity averages keyframes that are close to each other and reduces the ones that are considered "redundant" and can practically be replaced with an interpolated curve instead. You can find this if you select an fbx file that has animations on it, go to the "Animation" tab and set Anim. Compression to 'Keyframe Reduction':



This is not what the feature is actually supposed to do - the removed keyframes are a way to reduce the filesize, but it helps us regardless! If we increase or decrease the parameters for **Rotation Error**, **Position Error** and **Scale Error**, we'll change to what degree these corrections are applied:

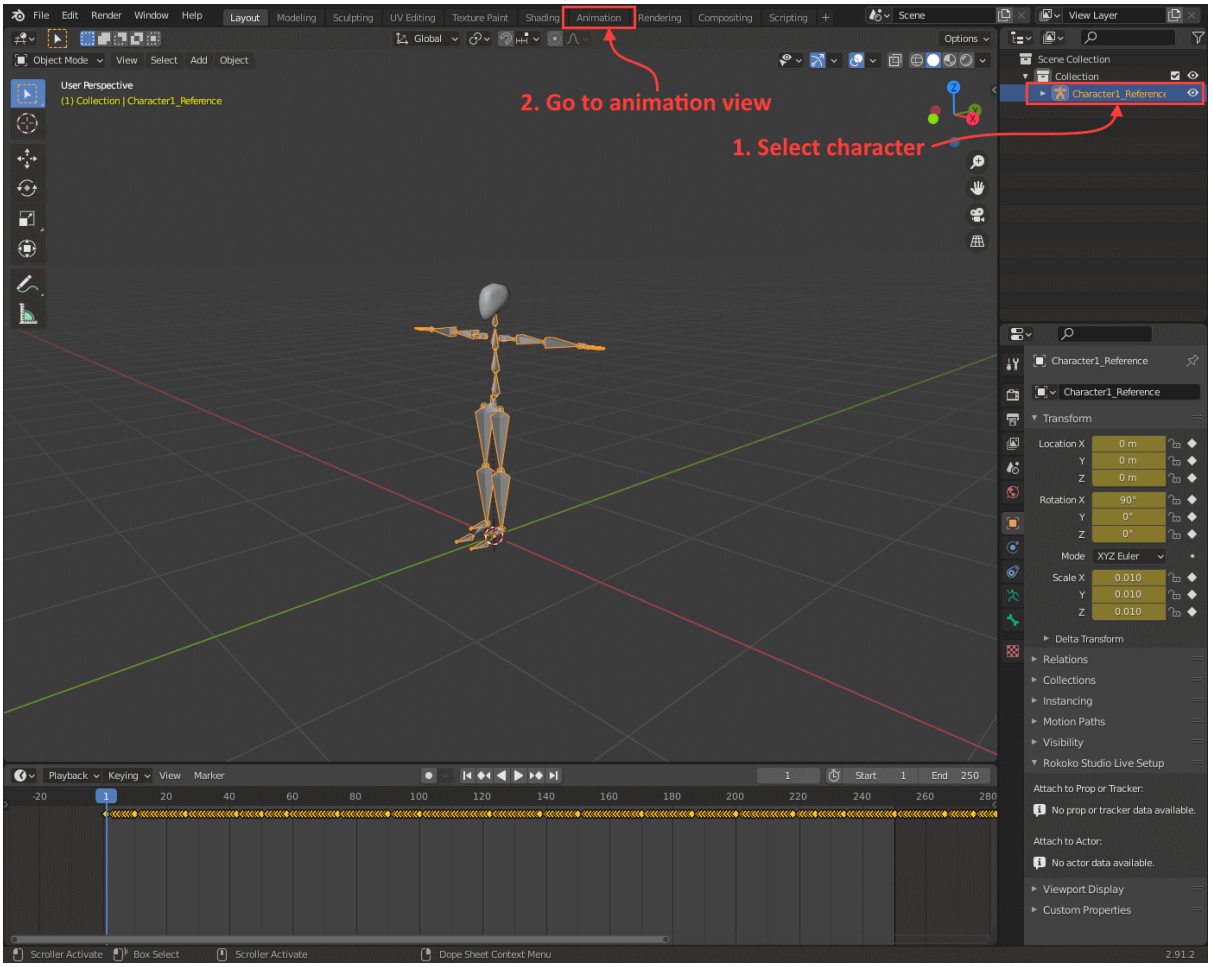


The values that are entered here are essentially the “delta” by which the evaluation is made - and if the rotation, position and scale differ less than this between each keyframe, then the keyframe is removed.

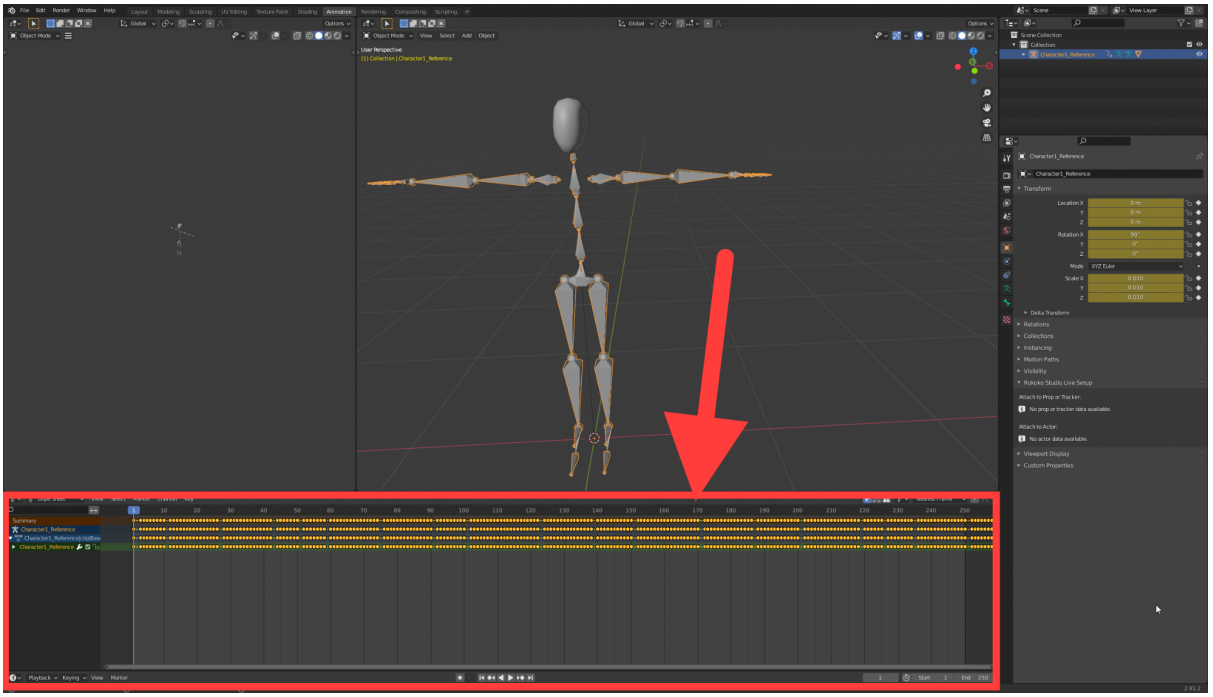
Though for some cases we’ll need to go further and apply our own keyframe reduction to a higher degree, or be more precise and increase reduction on specific bones, rather than the whole clip. Or if we are working with blendshape animations where we are just dealing with blend weights and not rotation, position or scale, we’ll have to move outside of Unity.

For such cases I recommend going to Blender and using a feature with a very dramatic name - **keyframe *decimation*!** Which is really just keyframe reduction.

Begin by opening Blender and removing the default cube, camera and light again, then import the fbx file. Next you will have to take a few steps to smoothen out your process, as I recommend not doing keyframe reduction on the entire animation at once - especially not if you have it exported with a very high number of frames per second. Here’s the first steps to get to the Animation view:



Once here, you should see the keyframe view at the bottom change, to show expanded information about each frame:



This is usually showing only a fraction of the keyframes in your animation, as it defaults to only showing 250 frames. You can scroll out to see everything and get an idea for where the last frame is, but I recommend pressing **CTRL+TAB** when in this keyframe view, to change to animation curves. Here is what the animation curves view looks like for a clip I have chosen, zoomed all the way out:

Apologies for the micro-text, it says “Start- and end-frames”.

As highlighted, in the lower right corner is where you can change the start and end frames for the animation. If you hover over the last keyframes and scroll you can zoom in to find the last frame that you can write as the end-frame, just to make sure the whole animation plays.

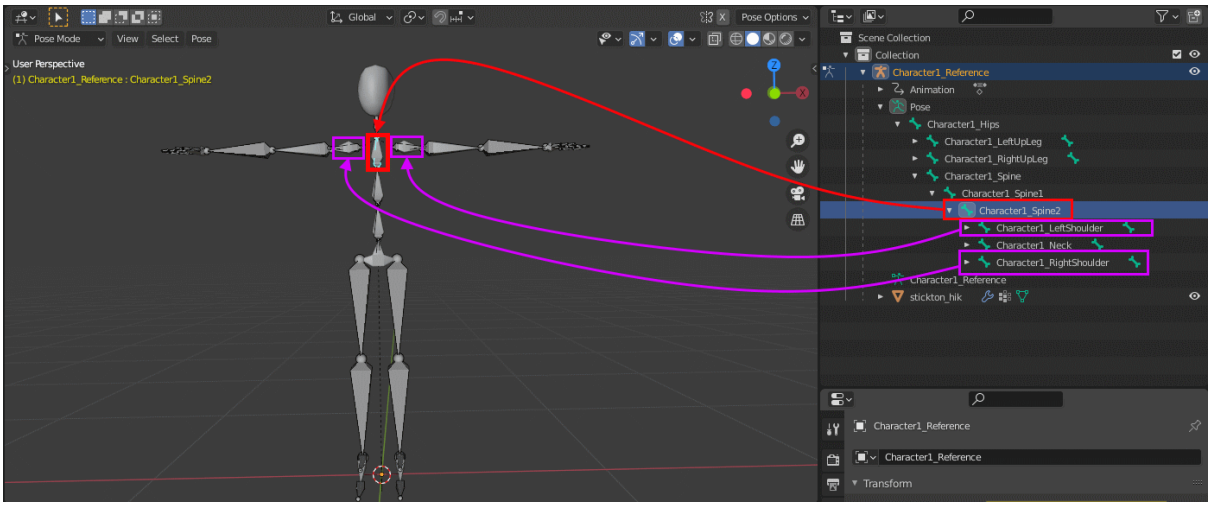
To preview the animation itself, there’s a few buttons in the middle of the animation view, down at the bottom, which are fairly self-explanatory:

You can basically play, skip to start/end, jump forwards/backwards between frames and (for some reason) play backwards. We won’t be using the two buttons on the left side, as they involve recording - we’re just here to clean up the mocap.

Before we get into reducing keyframes, it’s important to think about what we should reduce. While we *could* theoretically just select every single frame in the animation and reduce everything, most computers will have a very hard time doing this. Reducing keyframes across the board also gives the mocap more of a hand-animated feel - which can be great for stylized characters, but is more noticeable if you are going for a realistic feel.

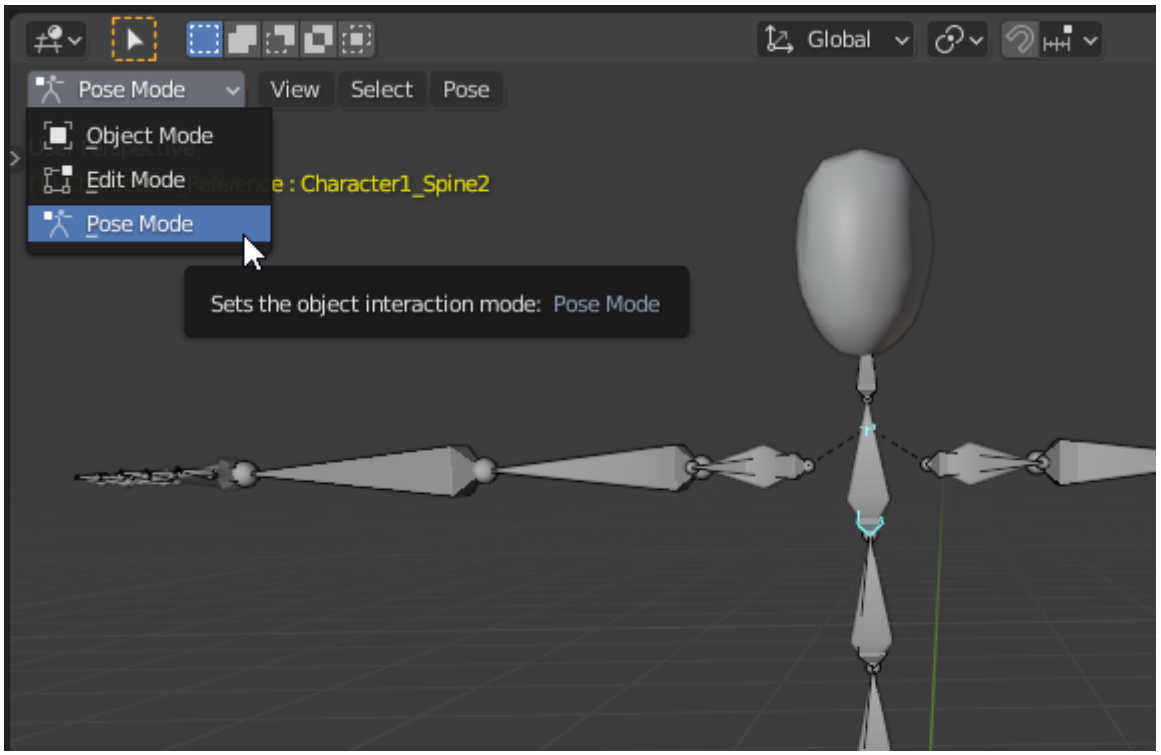
As for making the evaluation of what you should target, consider the fact that all motion propagates from the root and outwards, starting from the hip bone. So if your entire

character is experiencing jitter and you see that the hip bone is doing this motion as well, I would start with the hip bone and see how that affects the rest.
A more particular example would be - I see my hands doing a bit of a jittery motion, but the head of the character is not. That means this jitter is occurring somewhere along the arm and not in a bone that both the hand and head are sharing. To illustrate this, here's a comparative view of the character's skeleton, up to the point where the hands and head no longer share a parent bone:

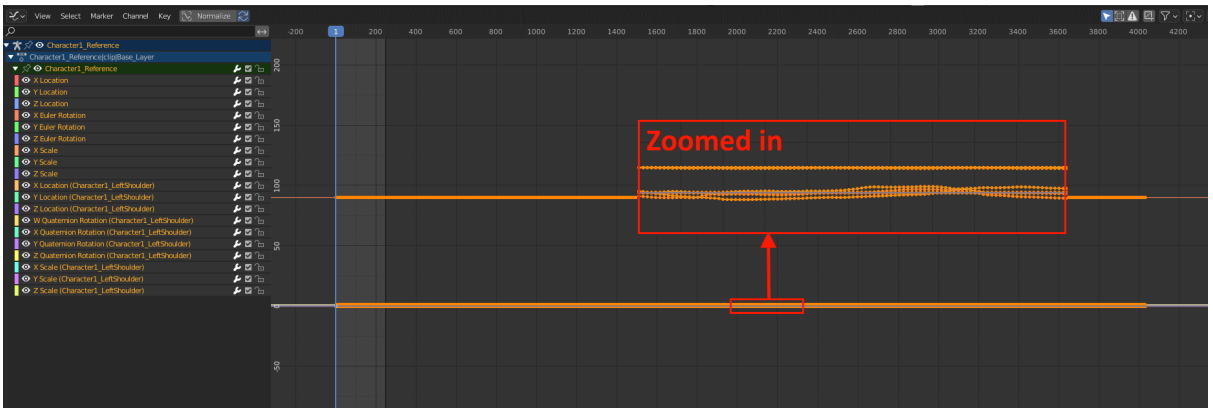


The red highlight is the last shared bone between the head and hands, meaning the purple bones are the first bones that could potentially be causing the jitter on the hands.

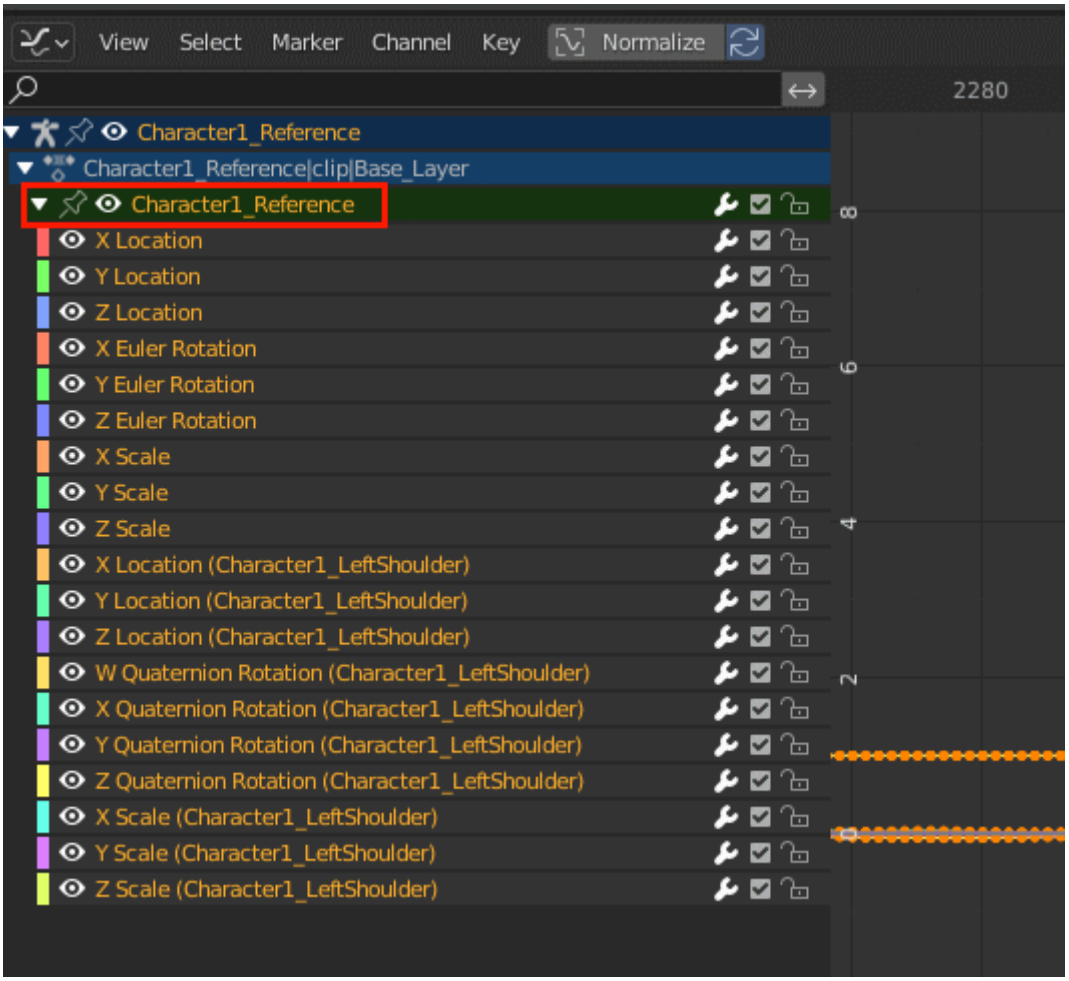
Before you can select what bones you want to target for keyframe reduction, make sure you are in "pose mode". It will say so in the upper left corner of the model view, just select it if something else is shown there:



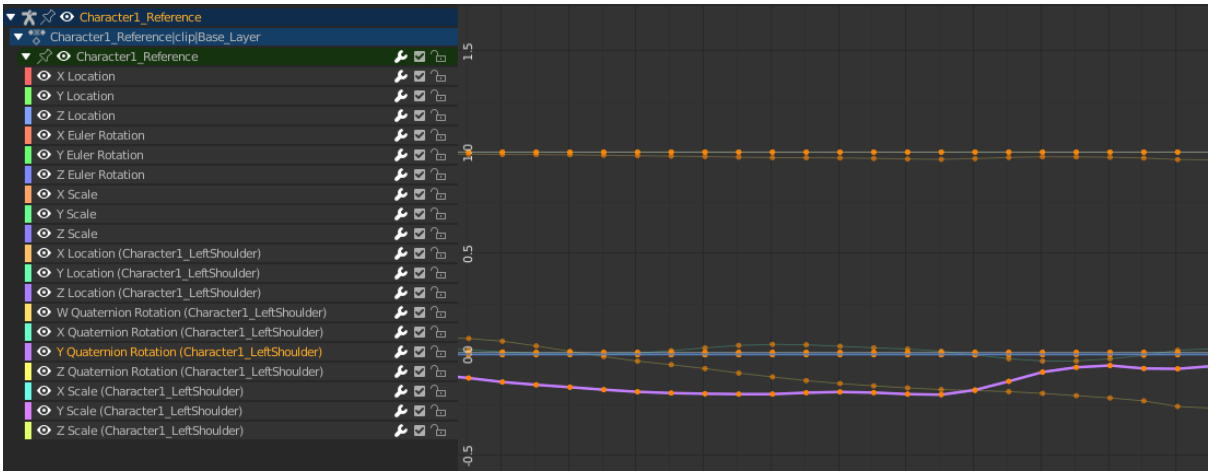
With this enabled, you can click directly on the bones to select them (shift-click for multiple). This will change what keyframes are visible in the animation view, to only display the data from the bone(s) selected. If you are zoomed all the way out, you might see the keyframes as an orange “smudge bar” rather than intelligible values. This does not mean something is wrong, it is simply due to how propagated motion (forward kinematics) works - the shoulder should not have large animation curves, as its movements are described relative to its parent bone, not in global space:



You may also notice that I have a list of all of the information that this bone’s keyframes contain. If you don’t see this list, make sure to click the little drop-down arrow on the green bar:

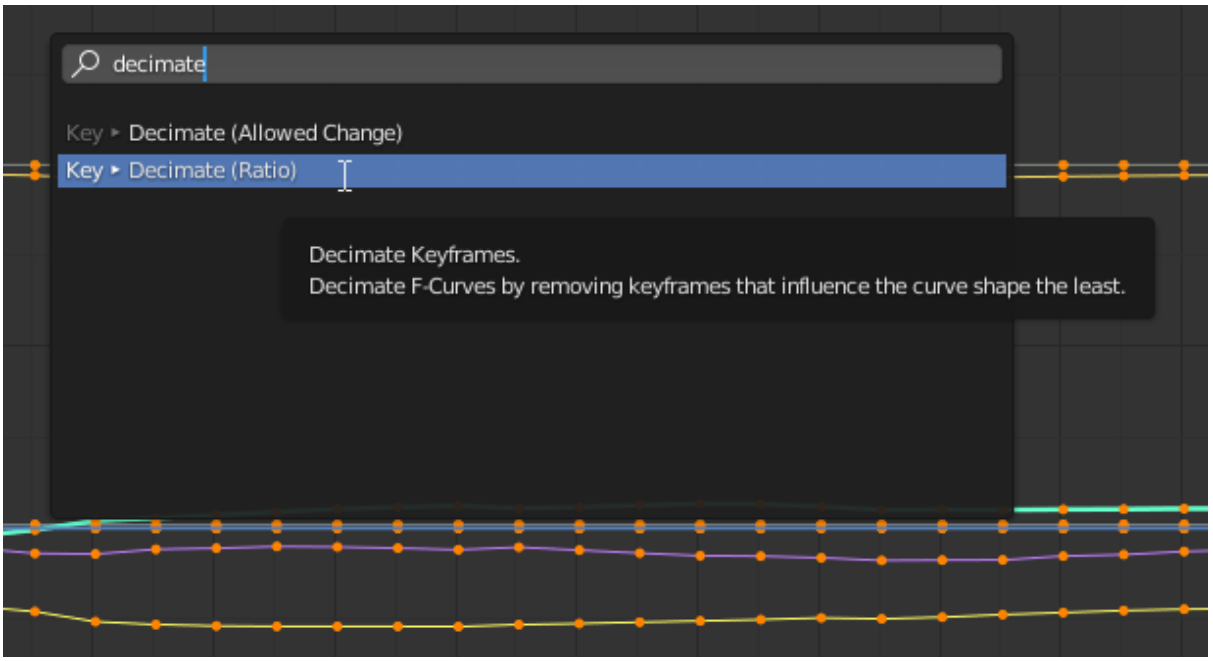


We can select/deselect which elements we are working with here if we want to be even more specific. If you click on each of these, it will highlight which one of the curves that represent this data:



What's interesting to note here, is that we are only actually animating the rotational values - as every property aside from Quaternion Rotations are showing as flat curves. Once again, that is due to how forward kinematics works, but it means that we can ignore everything else, because doing keyframe reduction to a flat curve isn't going to make it any smoother. The one exception is of course the hip bone, which can freely move however it likes, to move the rest of the skeleton.

Now for the actual keyframe reduction process! When you have selected which curves you want to reduce, **press F3** and search for "***Decimate***". You'll notice that there are two options, but I prefer the one using *Ratio*, as that is super easy to adjust, by moving the mouse right/left to increase/decrease the amount of reduction we are applying to the curves:



From here, it can be a little bit of a trial and error approach, so my advice is to iterate fast. In other words, try applying a *lot* of reduction to a bone to see if that did any changes further down the line. If no visible changes occurred, you can undo the reduction and move up to the next bone, until you find the one that is causing the jitter. However, be aware that it may be an accumulative behaviour - as in all of the bones in that chain may each be contributing a little bit. If that is the case, try marking several bones in a chain and apply a little bit of reduction, to see if that has a bigger impact on the end result. *Just be very selective with what you are reducing if you work with multiple bones - simply mark the rotations for reduction, if you do not have a super powerful computer to do this kind of work on.*

4.7.2 Doing pose corrections:

Just to clarify before I go into this - this section does not cover dynamic animation correction (like IK overrides) but rather how to modify the base animation files to make things line up in the animation clip itself. While I do want to dive into Unity's Animation Rigging package at some point, I think it is more important to ensure that your base clips are workable, before you attempt to do any kind of dynamic adjustment to them.

For once, I will actually refer most of this section to one of Sam's videos, which explains this process super well, as it is a topic that can be very difficult to communicate if not seen live:

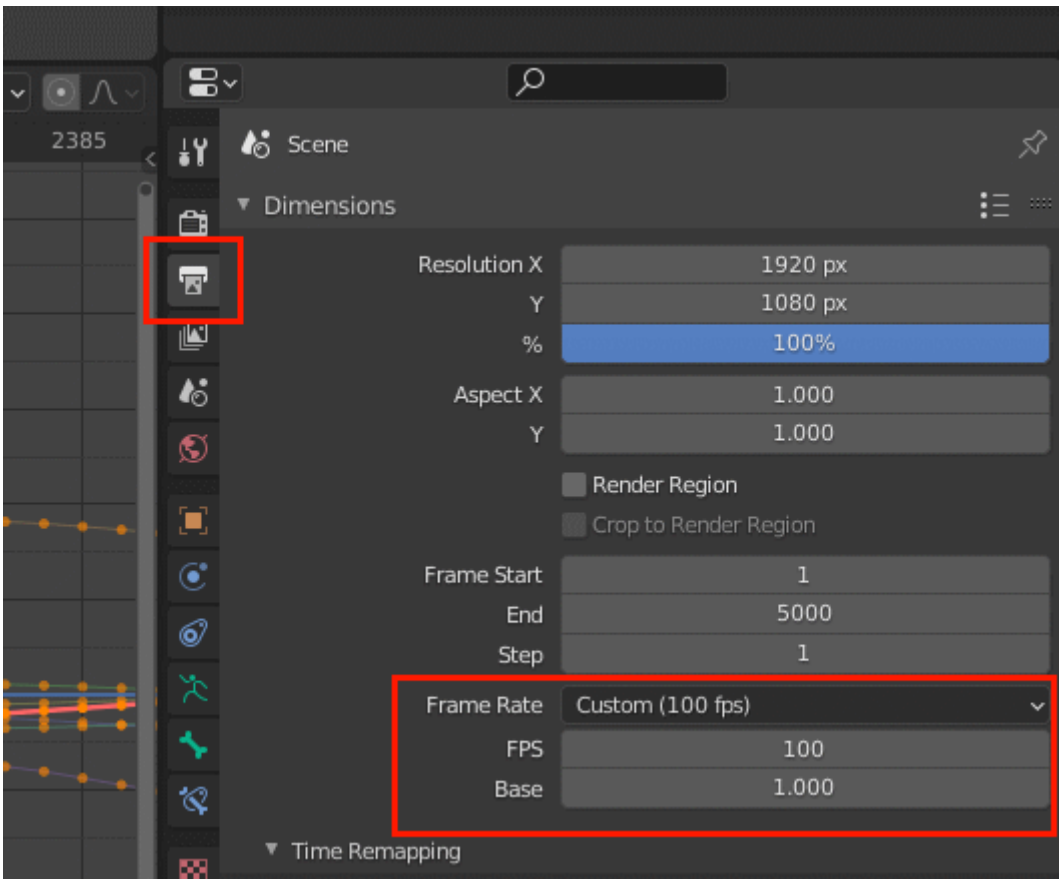
Easiest workflow for editing mocap in Blender - Rokoko Guide

To use this with a Unity-targeted workflow however, we'll need to get this work back out of Blender in a workable format. You should still import your animation file and character model, retarget to the character model (so that you have a proper reference on how this would play back on your character). However, when you are done with this process and want to put your animation back into Unity, you should only export the animation.

There are two steps that you should consider when doing this:

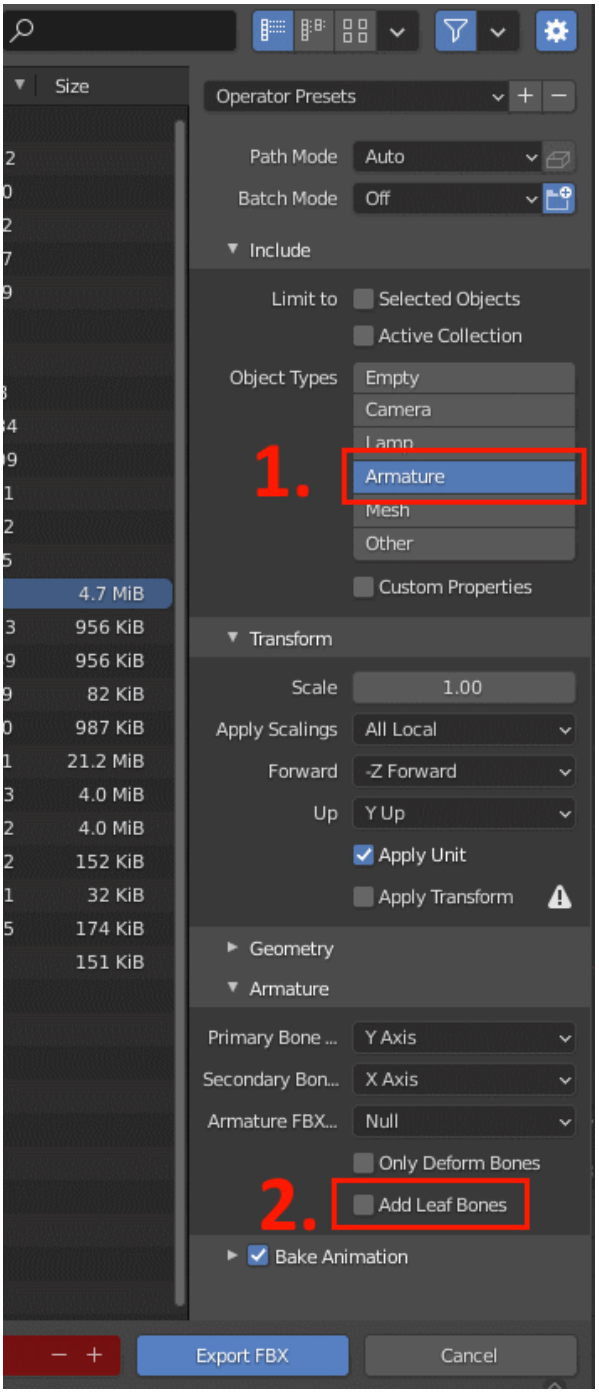
1. Make sure that the framerate still matches what you exported with Rokoko Studio.
2. Only export the skeleton (Blender calls this an armature).

Matching the framerate is very straightforward, but really important. If you look in the lower right region of your screen in Blender, there is a tab called "Output Properties". You want to click here and then set your framerate to the one you used in Rokoko Studio when you exported the animation, either from one of the values in the dropdown menu or by writing a custom value and pressing enter (which updates the animation, it doesn't happen by just typing the value):



Exporting the skeleton only is also pretty easy. All you have to do is the usual File > Export > FBX (.fbx), but then in the export settings window, do the following:

1. Deselect all Object Types except "*Armature*"
2. Make sure you aren't adding any leaf bones:



And finally, when you add the animation in Unity again, make sure you go through the process of adding this animation to a humanoid avatar, via Unity's fbx settings, as we previously went through.

5.0 Rounding off:

This covers the base fundamentals of getting your animations onto your character in Unity and playing them back. However, I can already hear you thinking; "but Dave, I have gone

through these instructions and all I have now is a wobbly viking that can circle around in front of a static camera, this looked much cooler on the article front-page!"

To which I say - **great**!

You now know enough to not just load and play Unity's starter assets bundle, but to modify it to your needs! I did not want to throw it at you right away because working with polished assets and complex animation systems is not representative of where you will start and what you can expect to run into. But now that you have graduated from beginner status, I highly recommend that you jump into it and start picking apart the third-person character controller. ***It has some great running animations and even jumping logic that you can easily reapply to your custom characters*** with the Humanoid Avatar system!

- Starter Assets - Third Person Character Controller | Essentials | Unity Asset Store

Here's how to get started with it (you can skip the first-person controller part):
- Starter Assets overview | Unity

6.0 Where to go next:

Now that you have a basic understanding of how to use these systems, you can begin to explore how they are used for different types of game mechanics. I have made a list of resources that I recommend you check out, which build upon what we have gone through here. I've also listed a few places that you can explore for additional resources, like animation clips, 3D assets etc.

6.1 Rokoko Discord Community

You've seen it linked in the footer on each page - a great place to start is to check out our Discord community! We have users from all kinds of places and that's where you'll find me hanging out as well - every weekday from 9am to 5pm (GMT+1) so please drop in and say hi! I'm always happy to help with Unity questions as well as general thoughts on game design, AI programming, audio design and most other areas of game dev - just ping Dave_Rokoko in the appropriate channels!

You can find me and the rest of the team at: https://discord.gg/rokoko

6.2 Online tutorials

With one of the best communities out there, Unity has an incredible amount of user made tutorials on just about anything you can think of. I've listed a few playlists and channels here to get you started:

- Jason Weiman: Animations with Layers in Unity3D - Unity Devs WATCH THIS

A great breakdown of animation layers in a more complex scenario.

- iHeartGameDev: Unity's Animation System - YouTube

An awesome walkthrough of Unity's animation systems in general, which covers some of the same ground as we have here, but gets into a few more examples and scenarios.

- Unity Learn: Unity Learn

The official Unity educational resource, covering all kinds of areas and skill sets. A great place to dive into other areas of expertise, outside of animation.

6.3 Asset resources

Need more animation clips, character models, sound effects etc.? Here's a few good suggestions:

- Motion Library in Rokoko Studio: Rokoko Motion Library

Has a bunch of neat animations. If you sort by price ascending, the first 100 assets on the list are free!

- Our Thursday stream at 19:00 GMT+1: Rokoko - YouTube

This is where Sam hangs out and does mocap requests. If you have something very specific in mind but do not have access to the hardware, this is the place to be. Subscribe to the channel to get notified, or keep an eye on Discord as we're hanging out before the stream starts as well.

- Many other places: Top 9 Game Asset Sites | Free 2D & 3D Game Assets

We wrote an article about good places to go for free game assets of all kinds, which you can find on our Insights blog!